CD-ROM付
Windows対応

10日でおぼえる C言語入門教室 第2版

坂下夕里 著

SE
SHOEISHA

本書の内容

誰でも身につけておきたいプログラミング言語といえば、今でも間違いなくCです！ 本書は、「本格的にC言語を学びたいが、まずは取り掛かりをつかみたい」「実際の動作イメージを体感しながら学びたい」という方のために、プログラミングの最初の一歩を提供するものです。

何はともあれあまり難しく考えず、自分の手でコードを打ち込み、自分の目でプログラムの動きを確認してみてください。

なぜ10日でおぼえられるのか?

本書は、実際に自分の手を動かしてサンプルプログラムを作りながら学ぶ、セミナー形式だからです。多くの入門書のように、「本を読んで内容の理解はできたけれども、自分でコードを1行も書けない」なんてことはありません。

【レッスンの構成】

1日数時限ずつこなしていけば、10日間で確実に「C言語で基本的なプログラムを作る力」が身につくよう、構成しています。

1. 今回作成する例題
その時限で作るプログラムの動作と、
そのレッスンのポイントを紹介します。

2. プログラムを作成する
手順に従って、プログラムを自分で作ってみます。

3. 解説
作成したプログラムに則して、C言語の基礎を学びます。

10日でおぼえる C言語入門教室

第2版

坂下夕里 著

SE
SHOEISHA

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願いしております。下記項目をお読みいただき、手順に従ってお問い合わせください。

●お問い合わせの前に

弊社 Web サイトの「正誤表」や「出版物 Q&A」をご確認ください。これまでに判明した正誤や追加情報、過去のお問い合わせへの回答（FAQ）、的確なお問い合わせ方法などが掲載されています。

正誤表　http://www.seshop.com/book/errata/
出版物 Q&A　http://www.seshop.com/book/qa/

●ご質問方法

弊社 Web サイトの書籍専用質問フォーム（http://www.seshop.com/book/qa/）をご利用ください（お電話や電子メールによるお問い合わせについては、原則としてお受けしておりません）。

※質問専用シートのお取り寄せについて
Web サイトにアクセスする手段をお持ちでない方は、ご氏名、ご送付先（ご住所／郵便番号／電話番号または FAX 番号／電子メールアドレス）および「質問専用シート送付希望」と明記のうえ、電子メール（qaform@shoeisha.com）、FAX、郵便（80 円切手同封）のいずれかにて“編集部読者サポート係”までお申し込みください。お申し込みの手段によって、折り返し質問シートをお送りいたします。シートに必要事項を漏れなく記入し、“編集部読者サポート係”まで FAX または郵便にてご返送ください。

●回答について

回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

●ご質問に際してのご注意

本書の対象を越えるもの、記述個所を特定されていないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、あらかじめご了承ください。

●郵便物送付先および FAX 番号

送付先住所　〒 160-0006 東京都新宿区舟町 5
FAX 番号　03-5362-3818
宛先　（株）翔泳社 出版局 編集部読者サポート係

※本書に記載された URL 等は予告なく変更される場合があります。
※本書の出版にあたっては正確な記述に努めましたが、著者および出版社のいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。
※本書に掲載されているサンプルプログラム、および実行結果を示した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。

ようこそ C言語入門教室へ！

本書のすべてのレッスンをおえるころには、C言語の基本をマスターして、一人で基礎的なプログラムが書けるようになっているはずです。

コンピュータに仕事をさせる「プログラム」は「プログラム言語」を使って書きます。そのプログラム言語は、コンピュータが開発されてから現在まで数多く作られてきました。

「プログラム言語」というと、最近はJavaやPerlなど、インターネット上のアプリケーションを開発するために使われる言語が有名です。プログラムに縁のない人でも聞いたことがあると思います。

一方、プログラムを作る仕事に従事したり、学校で情報処理を専攻したりする人……いわゆる玄人が「基本言語」として習う言語の代表がC言語です。こうしたいわゆる玄人でなければあまり目にすることも気にすることもないように思えますが、C言語やそれを元に開発された言語（C++やVisual C++）を使うと、コンピュータ上の色々なソフトウェア、例えばメールソフトやWebブラウザなどを作ることができます。

本書はC言語を基礎から学ぶ入門書です。

どのプログラム言語でもいえることですが、プログラム言語にはそれぞれ数え切れないほどの文法が用意されています。これを全ておぼえるのは大変です。よって本書では、各日各時限ごとに作りたいプログラムを決めて、そのプログラムで使わなければならない文法や必要となる関数のみを効率よく学習します。基本的には第0日から順にレベルアップをしているので、最後の第10日まで学習すれば、C言語の基本はほぼマスターしているはずです。

各時限で作成したプログラムでは使っていないけれど、C言語の文法上知っておくべき重要な内容が出てきた場合は、そのつど説明を入れるか、または各時限がおわった最後に説明してあります。

本書で作るプログラムのほとんどはゲームプログラムです。簡単なものからはじめていきます。最後は、さすがにメールソフトやWebブラウザなどのプログラムを作るまではいきませんが、ちょっと複雑なゲームを作るところまで学習します。

本書で作る楽しみと遊ぶ楽しみの両方を味わいながら、学習を進めましょう。

本書は、「10日でおぼえるC言語入門教室」を加筆した改訂版になります。改訂するにあたり、より多くのゲームプログラムが作成できるようにしました。また、学習環境をより楽に整えるため、使用するコンパイラをMinGW日本版に変更しました。

2009年7月　坂下夕里

CONTENTS

COLUMN

付属CD-ROMをご使用の前に

**付属CD-ROMを使用する際の注意等です。
CD-ROM中のファイル「readme.txt」もあわせてお読みください。**

●付属CD-ROMについて

本書で解説している例題のサンプルソース一式、およびMinGW日本版は、付属のCD-ROMに収録しています。適宜、必要ファイルをハードディスクにコピーしてご使用ください。またそのファイルは読み取り専用になっています。かならず読み取り専用属性をはずしてからご使用ください。

MinGW日本版のインストール方法については、第0日を参照してください。

●免責事項について

CD-ROMに収録されたファイルは、通常の運用においてはなんら問題のないことを編集部および著者は確認していますが、万一運用の結果、いかなる損害が発生したとしても、著者および翔泳社はいかなる責任も負いません。すべて自己責任においてお使いください。

●CD-ROMのテスト環境

CD-ROMは以下の環境で正常に動作することを確認しました。

・Windows XP（Professional）
・Windows Vista（Home Premium）

●著作権等について

本書に収録したソースコードの著作権は、著者および(株)翔泳社が所有します。個人で使用する以外には使うことはできません。許可なくネットワークなどへの配布はできません。個人的に使用する場合には、ソースコードの改変や流用は自由です。商用利用に関しては（株）翔泳社へご一報ください。

株式会社翔泳社　編集部

第 **0** 日

オリエンテーション

C言語の基礎知識

プログラムについて

C言語でのプログラミング環境を整えよう

**本書は、C言語の初心者だけでなく、コンピュータプログラム自体、はじめて挑戦する方を対象とした、C言語の入門書です。
専門書を最後まで読み切ることは多大な気力を必要としますが、本書ではできるだけ途中で挫折しないよう、プログラムを作る楽しみを味わってもらうために、いろいろなゲームプログラムを作りながら学習を進めていきます。
さっそく何か簡単なゲームプログラムから……とはじめたいところですが、何事にも準備が必要です。本日はC言語の学習に入る前に知っておきたい基礎知識と、「プログラムって何？」という人のために、コンピュータプログラムについての基礎を学習します。また最後に、PC上でC言語のプログラムを動かすための環境を用意します。**

C言語の基礎知識

最初にC言語についての基礎を学びます。
まだ、プログラムは作りません。最初から気合を入れて説明だけを読むと、途中で力尽きて、本が本棚の飾りになってしまうことがあります。身におぼえのある人もいるでしょう。この時間の説明は力を抜いて、楽に読み流してください。
ではさっそく、C言語の世界へ第一歩を踏み出しましょう！

コンピュータプログラムとは何か

コンピュータは人間が指示した命令のとおりに動きます。この命令の手順を書いたものが、コンピュータプログラムです。以降は単にプログラムと呼びましょう。

命令の手順の書き方は、いくつかの種類があります。それがプログラム言語の違いです。COBOL、Fortran、C、BASIC、Java、Perlなど、コンピュータの普及とともに、さまざまな言語が開発されてきました。

例えば、コンピュータに、

① ああして
② こうして
③ こうして
④ これをやってね

といった命令の手順をプログラムとして書く場合、C言語で書くこともできるし、Javaで書くこともできます。どの言語を使って書いても、コンピュータが実行する処理にほとんどかわりはありません。

●CとJavaのプログラム

Cプログラム

```
#include <stdio.h>

int main() {
    int a = 2, b = 3;
    int c, d;

    c = a + b;
    d = a * b;
    printf("%d %d", c, d);
    return 0;
}
```

Javaプログラム

```
public class sample {
    public static void main(String[] args) {
        int a = 2, b = 3;
        int c, d;

        c = a + b;
        d = a * b;
        System.out.println(c + " " + d);
    }
}
```

実行結果

5 6 ←→ 5 6

同じ！

プログラム言語の違いは、日本語や英語、フランス語といった言語の違いと思ってもらってかまいません。それぞれの言語で意味する内容は同じでも、単語や文法が異なります。

プログラム言語の世界では、コンピュータで行う処理は同じでも、コンピュータに指示する命令文やその書き方が言語によって異なります。

書きやすくて読みやすいC言語

C言語の主な特徴は次の4点です。

・比較的コンピュータに近い言語である
→細かいレベルまでプログラム制御が可能

・プログラムの構造は関数（プログラムのあつまり）を基本としている

・ファイルの入出力や文字列操作などをひとつの関数として扱える
→プログラムが書きやすく、読みやすい

・プログラムのソースを他のOSに移植することが容易である
→他の環境で使うときにプログラムの変更がない（少ない）

C言語の一番の特徴は「プログラムが書きやすく、読みやすい！」ということに尽きます。

今ではさまざまなプログラム言語が存在していますが、どの言語を選択してプログラムを書くかは、目的次第です。「こういう目的のプログラムを作るには、この言語を使え！」と必ずきまっているわけではありませんが、プログラム言語にはそれぞれの目的に対して向き不向きがあるので、自然と用途はきまってきます。

例えば、FortranやBASICは科学技術計算用、JavaやPerlは主にインターネットのWebアプリケーション用として使われます。

一方、C言語は目的にとらわれず、何にでも利用できる汎用プログラム言語として有名です。実際、C言語で書かれたプログラムは何にでも使われています。

Windowsのソフトウェアにも C言語で書かれたものがあります。Windowsのソフトウェアの多くは、GUI（グラフィックユーザインタフェース）といって視覚的な操作を行いますが、これにはC言語やC言語をもとにしたC ++言語で書かれたものが多くあります。

また、C言語はコンパイラ型言語*1で動作が速いため、Webアプリケーションとしても使用されています。

ヒント
*1：コンパイラ型言語については、あとで詳しく説明します。

プログラムを作って実行するには？

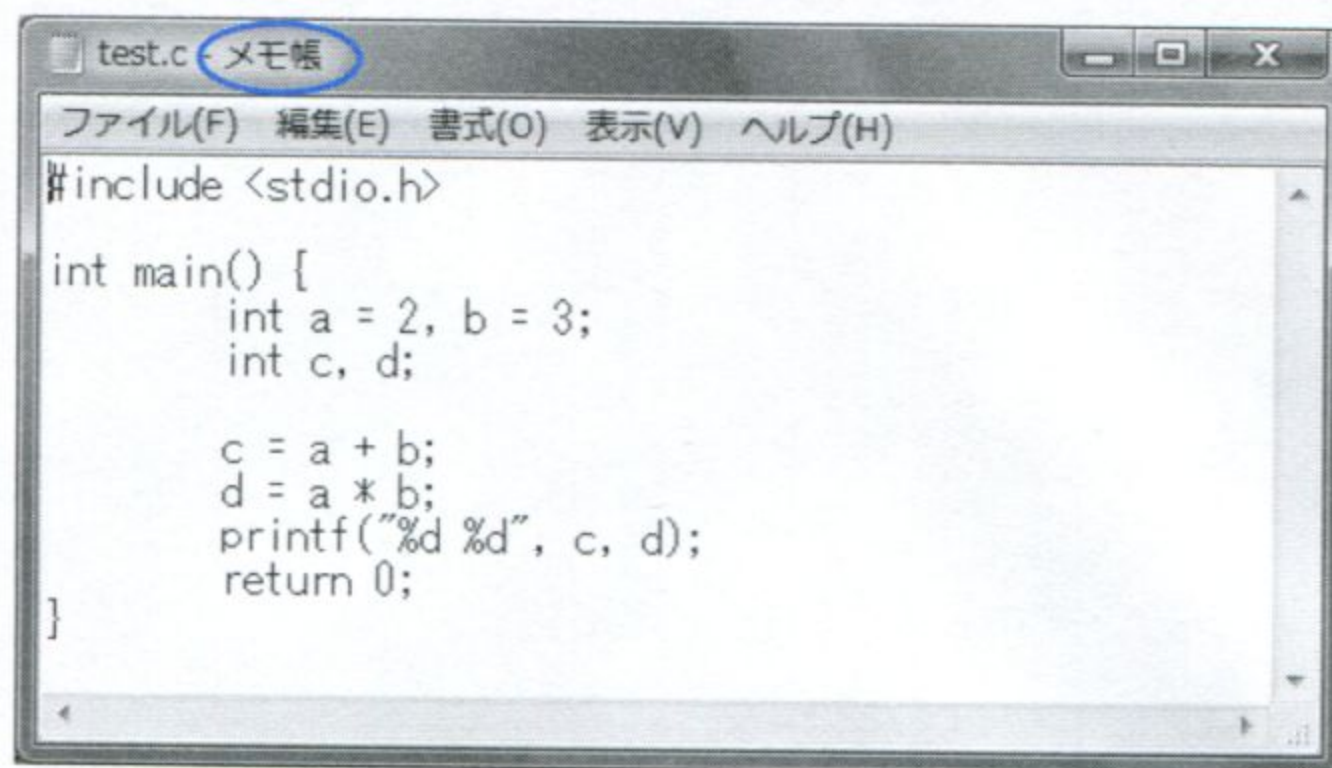
プログラムはコンピュータに対する命令手順書ですから、基本的には人間が普通のテキストエディタ（メモ帳や秀丸エディタなど）を利用して書きます。プログラムをテキストエディタ[*2]で書き、保存したものが、プログラムファイルです。

ヒント
*2：テキストエディタとは、文字のみで何もレイアウト情報を持たないテキストファイルを編集・作成するソフトウェアです。

●メモ帳で書いたC言語のプログラムファイル

test.c - メモ帳

ファイル(F)　編集(E)　書式(O)　表示(V)　ヘルプ(H)

```
#include <stdio.h>

int main() {
        int a = 2, b = 3;
        int c, d;

        c = a + b;
        d = a * b;
        printf("%d %d", c, d);
        return 0;
}
```

プログラムはテキストエディタで書ける

プログラムファイルのままでは、コンピュータにとっては、ただのテキストファイルです。この状態では、コンピュータは、「命令手順書」として理解することはできません。

コンピュータに理解してもらうためには、このプログラムファイルから実行ファイル（拡張子[*3]が「.exe」のファイル）を作り出す必要があります。

ヒント
*3：大抵のファイルは「sample.txt」のように「.」で区切った名前がつけられています。この「.」からうしろを拡張子と呼びます。拡張子は、ファイルの種類を識別するためのものです。

①実行ファイルとは？

実行ファイルとは、一般的に「ソフト」や「ソフトウェア」と呼ばれるものです。例えば、Webブラウザの代表的なソフトウェアにInternet Explorerがあります。タスクバーにあるInternet Explorerのアイコンを右クリックして、「プロパティ」を出してみましょう。

ヒント
*4：Windows XPの場合は、「スタート」－「すべてのプログラム」－「Internet Explorer」を右クリックして、「プロパティ」を出します。

● Internet Explorerのプロパティ

開く(O)
ファイルの場所を開く(I)
管理者として実行(A)...
[スタート] メニューにアイコンを追加(I)
クイック起動に追加
送る(N)
切り取り(T)
コピー(C)
ショートカットの作成(S)
削除(D)
名前の変更(M)
プロパティ(R)

右クリック

Internet Explorer ブラウザの起動のプロパティ

全般　ショートカット　互換性　セキュリティ　詳細

Internet Explorer ブラウザの起動

種類：アプリケーション
場所：Internet Explorer
リンク先(T)：“C:¥Program Files¥Internet Explorer¥iexplore.exe”
作業フォルダ(S)：%HOMEDRIVE%%HOMEPATH%
ショートカットキー(K)：なし
実行時の大きさ(R)：通常のウィンドウ
コメント(O)：インターネット上の情報および Web サイトを検索して表示し
ファイルの場所を開く(F)　アイコンの変更(C)...　詳細設定(D)...
OK　キャンセル　適用(A)

Internet Explorerの「プロパティ」

「ショートカット」タブにある「リンク先」を見ると、「なんたらかんたら¥iexplore.exe」になっているはずです。つまり、Internet Explorerというソフトの実体が、「iexplore.exe」という実行ファイルであることがわかります。

筆者の環境では、この実行ファイルは、

```
C:¥Program Files¥Internet Explorer¥iexplore.exe
```

にあります。このファイルを探し出し、テキストエディタで中を見ようとすると大変です。非常に大きいファイルですし、仮に中を見ても何が書いてあるのか私達にはわかりません。しかし、コンピュータはこの「何が書いてあるか人間にはわからない」ファイルの内容を理解することができます。なぜなら、「人間にはわからないが、コンピュータ（機械）にはわかる言語」、すなわち「機械語」で書かれているからです。

実行ファイルとは、プログラムファイルを機械語に翻訳したファイル、ということになります。実行ファイルは機械語で書かれていることを、おぼえておいてください。

②自分で作ったプログラムを実行ファイルにするには？

ソフトウェアは、必ず誰かが作ったものです。WebブラウザのInternet Explorerも、誰かが「Webサイトを閲覧するプログラム」を書いて、それを機械語に変換して作り出した実行ファイルです。読者のみなさんがこれから作るプログラムも、機械語に変換して実行ファイルを作り出せば、立派な「ソフトウェア」になります。

では、プログラムファイルから実行ファイルを作り出すには、どうしたらよいでしょうか[*5]？

ヒント

*5：各言語で書かれたプログラムの内容を「ソース」、または「ソースコード」といいます。単にプログラムという場合もありますが、ソースを機械語に変換した実行ファイルも「プログラム」と呼ぶ場合があるので、注意が必要です。
本書ではこれ以降、プログラム言語で書かれたファイルを「プログラム（ソース）ファイル」と呼び、プログラムファイルを機械語に変換した実行ファイルを「実行ファイル」と呼んで使い分けます。

●実行ファイルができあがるまでの流れ

1. プログラムファイル test.c

```
#include <stdio.h>

int main() {
    int a = 2, b = 3;
    int c, d;

    c = a + b;
    d = a * b;
    printf("%d %d", c, d);
    return 0;
}
```


1. プログラムファイルを作成する
2. プログラムファイルを機械語に変換する（この作業をコンパイルと呼びます。できたファイルはオブジェクトファイルといいます）
3. オブジェクトファイルと、プログラムを実行するのに必要な他のファイルとを結合する（この作業をリンクと呼び、実行ファイルを作り出します）
4. 実行ファイルのできあがり！

1の作業は人間が行います。2と3の作業は、Cコンパイラと呼ばれるソフトウェアが行ってくれます。つまり、C言語で書かれたプログラムファイルから実行ファイルを作成するには、C言語の知識とCコンパイラのソフトウェアが必要なのです。

C言語の知識については、これから本書で学びます。Cコンパイラの入手方法は、のちほど説明します。

まとめ

プログラムは、コンピュータに何らかの作業をさせる「作業手順書」です。これをコンピュータが理解できる「機械語」に変換することより、ソフトウェアを作り出すことができます。

皆さんがいつも使っているメールソフトやWebブラウザなども、もともとは誰かが書いたプログラムから作られています。プログラムを書くときの言葉がプログラム言語であり、C言語もそのうちのひとつです。本書はC言語を基礎から学んでいきますが、学習を進めれば、将来はWindowsのソフトウェアも作れるようになるかもしれません。

COLUMN

インタープリタ型言語とコンパイラ型言語について

プログラム言語にはさまざまな種類がありますが、大きくはインタープリタ型言語とコンパイラ型言語の2つに分類されます。
ソースコードをコンピュータが実行できる形式（オブジェクトコードといいます）に変換しながら、そのプログラムを実行するのがインタープリタ型言語です。PerlやPHPが、インタープリタ型言語の代表です。
コンパイラ型言語は、実行する命令のみをあらかじめ機械語に変換してあるので、インタープリタ型言語に比べて実行速度が速くなります。

●コンパイラ型言語とインタープリタ型言語の実行方法

コンパイラ型言語

```
#include <stdio.h>

int main() {
    int a = 2, b = 3;
    int c, d;

    c = a + b;
    d = a * b;
    printf("%d %d", c, d);
    return 0;
}
```


インタープリタ型言語

```
#!/usr/local/bin/perl

$a = 2;
$b = 3;

$c = $a + $b;
$d = $a * $b;
print "$c $d";
```


C言語はコンパイラ型言語なので、インタープリタ型言語であるPerlなどに比べると、同じ処理でも格段に実行速度が速くなります。
ここまで紹介してきたC言語の特徴をみると、プログラムの実行は速いし、ソースも書きやすい、移植性も汎用性もある……とよいこと尽くめなので「だったら他の言語はいらないんじゃないの？」と思うかもしれません。しかし、やはりどの言語にもメリットとデメリットが存在します。
ソースが書きやすいといっても、一般的にはインタープリタ型言語の方が簡単です。先にPerlやPHPを学習している人は、C言語やJavaでのデータの扱いを面倒だと思うでしょう。
C言語は、「万能で便利だけど、インタープリタ型言語よりは簡単ではない言語」とおぼえておきましょう。

プログラムについて

オリエンテーションが終了すれば、実際に本書に掲載しているサンプルコードを入力しながら、自分でプログラムを書いて学習する予定です。書き写すことに慣れてくると、なんとなく「プログラムとはこういうものなのか」とわかってくると思います。
ですが、プログラムを書く前にある程度の予備知識を持っていると、さらに理解しやすくなります。ここでは、プログラムを書きはじめる前にあらかじめ知っておきたいことを、いくつか紹介します。

プログラムのルール

プログラムを書くためには、言語ごとにきめられた書き方をしなければなりません。C言語のプログラムを書くには、C言語独自のルールに従ってソースコードを書きます。

場合分けする処理や繰り返しを行う処理など、文法上のルールは第１日以降の学習で、そのつど説明します。

まずはプログラムを書くときの最低限のルールを、ここで理解しておきましょう。

①プログラムは命令の手順を書くもの

プログラムでは、実行したい順に命令を書きます。

例えば、「1」「2」「3」を順に表示したい場合、

```
printf("1");
printf("2");
printf("3");
```

と書きます。

②ひとつの命令の最後には、必ず「;」をつける

先の①の例でいうと、「printf("1");」「printf("2");」「printf("3");」が、それぞれひとつの命令になります。

③プログラムは基本的に半角英数文字で記述する

プログラムは半角英数文字で記述します。スペース（空白）も半角が基本です。

```
ｉｎｔ　ｍａｉｎ（）　｛ ←――― 全角文字なので間違い

int main() { ←――― 正しい
```

④空白、改行、タブを利用する

プログラムを見やすくするために、半角スペースや改行、タブ（［Tab］キー）*6を利用することがあります。このとき、誤って半角スペースのかわりに全角スペースを入れないように注意してください。

⑤予約語を他の目的に利用してはいけない

予約語*7と呼ばれる単語は、Cコンパイラが特定の目的に使用するものです。予約語と同じ単語を、プログラムの中で変数名や関数名に使ってはいけません。

⑥大文字と小文字は区別する

C言語のプログラムには、必ず「main()」という記述が出てきますが、これを「Main()」や「MAIN()」と書いてはいけません。

⑦変数名や関数名には半角英数文字を使う

プログラムの中で使用する変数名や関数名は、半角英数文字を使って自分で自由につけることができます*8。しかし、全角文字や半角カナ文字は使えません。

⑧「//」から行のおわりまでは、コメント文とみなされる

「//」から行のおわりまでは、コメント文になります。「/* ～ */」で囲まれた部分も同様です。後者の書き方をすると、複数行をコメント文にできます。

なお、コメント部分はプログラムが実行されないので、全角文字を使ってもかまいません。

以上がC言語のプログラムを書くときの、最低限のルールです。

●ルールに従ってC言語で書かれたプログラム例

```
#include <stdio.h>

// サンプルプログラム          ← 1行コメント文
int main() {
    int count; // カウント数    ← 行の途中からコメント文
    char str[10]; /* 文字列変数 */  ←

    /* ここに処理を書きます     ← 複数行コメント文
       コメント文です */        ←
    return 0;
}
```

⑨ファイル名の命名規則について

最後に、C言語のプログラムファイルの命名に関する規則を紹介します。

先に述べましたが、拡張子はファイルの種類を識別するためのものです。C言語で書いたプログラムファイルの拡張子は、「.c」です。ファイル名は半角英数文字を使って自由に

ヒント

*6：タブを半角スペースでいくつ分にするかは、テキストエディタの設定によります。一般的には4か8が妥当です。

ヒント

*7：予約語の一覧は次ページで説明します。

ヒント

*8：⑥の法則はここでも反映されます。自分で作った「amari」という変数名を「Amari」と書いたら、別のものになってしまいます。

つけることができます。

C言語のプログラムを機械語に翻訳（コンパイル）すると、拡張子が「.exe」の実行ファイルが作成されます。

●C言語プログラムファイル名と実行ファイル名の例

プログラムファイル名	コンパイル	実行ファイル名
sample.c	──→	sample.exe
outputTest.c	──→	outputText.exe
2-2.c	──→	2-2.exe

まとめ

C言語のプログラムを書くときのルールについて学習しましたが、まだ実際にプログラムを書いていないので、ピンとこないでしょう。

これらのルールは、本書を読み進めながらソースを写しているうちに自然と身についてくることですが、知っておくと、より早く身につけることができるので、先に紹介しました。少しだけプログラムを書くことに慣れてきた頃に、もう一度戻って確認してみるとよいでしょう。

COLUMN

C言語の予約語

C言語では予約語と呼ばれる単語が用意されています。
基本的にC言語のプログラムでは、プログラムを作成する人が使いたい変数名や関数名を、自由につけることができます。しかし、予約語はC言語のプログラムを実行するためのシステムで使うので、予約語と同じ名前の変数や関数は使えません。

●C言語の予約語一覧

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

C言語でのプログラミング環境を整えよう

実際の講習ならあくびが出てくる頃なので、そろそろ手を動かしてみましょう。C言語で書いたプログラムを実行するには、Cコンパイラと呼ばれるソフトウェアが必要です。
「コンパイラ」と名前がついていると非常に大規模なものに聞こえますが、インストール方法や設定は非常にシンプルで、他のソフトウェアと比べてそれほど難しいことはありません。本書では、コンパイラを付属CD-ROMからコピーして使用できるようになっています。あとは、多少の設定を行えば、C言語のプログラムを書く準備は完了です。

Cコンパイラ環境を整える

Cコンパイラを入手して、C言語のプログラムを実行する環境を整えましょう。

本書ではWindows VistaとWindows XPの場合について説明します。なお、以下ではWindows Vistaでの設定画面を用いて説明しています。Windows XPをご利用の場合は、「ヒント」欄も参考にしてください。

Windowsで使用できるCコンパイラは複数存在します。本書では無料で配布されているCコンパイラのひとつである「MinGW日本版」を使用します。

MinGW*9（Minimalist GNU For Windowsの略）はWindows用のフリー開発環境で、それに日本語を使用できるようにしたものが、MinGW日本版です。

Cコンパイラをインストールする際は、コンピュータの管理者権限を持つアカウントで行ってください。

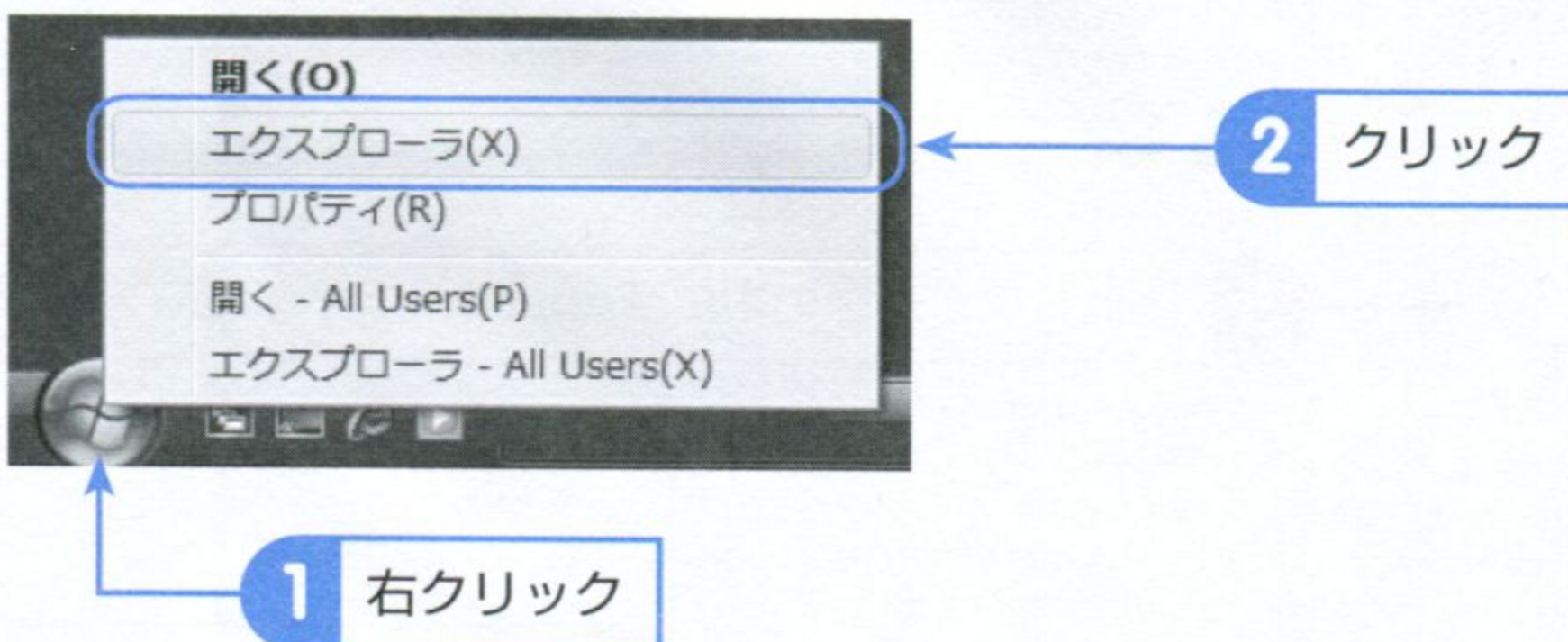
①コンパイラをPC上にコピーする

本書付属CD-ROMに収録してあるMinGW日本版を、ご使用のPC上にコピーします。

PC上のファイルやフォルダの管理はエクスプローラを使います。まず、タスクバーの「スタート」ボタンを右クリックして、エクスプローラを立ち上げます*10。

開く(O)
エクスプローラ(X)
プロパティ(R)
開く - All Users(P)
エクスプローラ - All Users(X)

2 クリック

1 右クリック

ヒント

*9：MinGWはgccというコンパイラをWindows上に実装したものです。gcc（GNU Compiler Collection）は、GNUプロジェクトによるUNIX系OSで使うフリーなコンパイラです。
またGNU（グニュー）とは、UNIXとの互換フリーソフトウェアの開発プロジェクトの総称です。

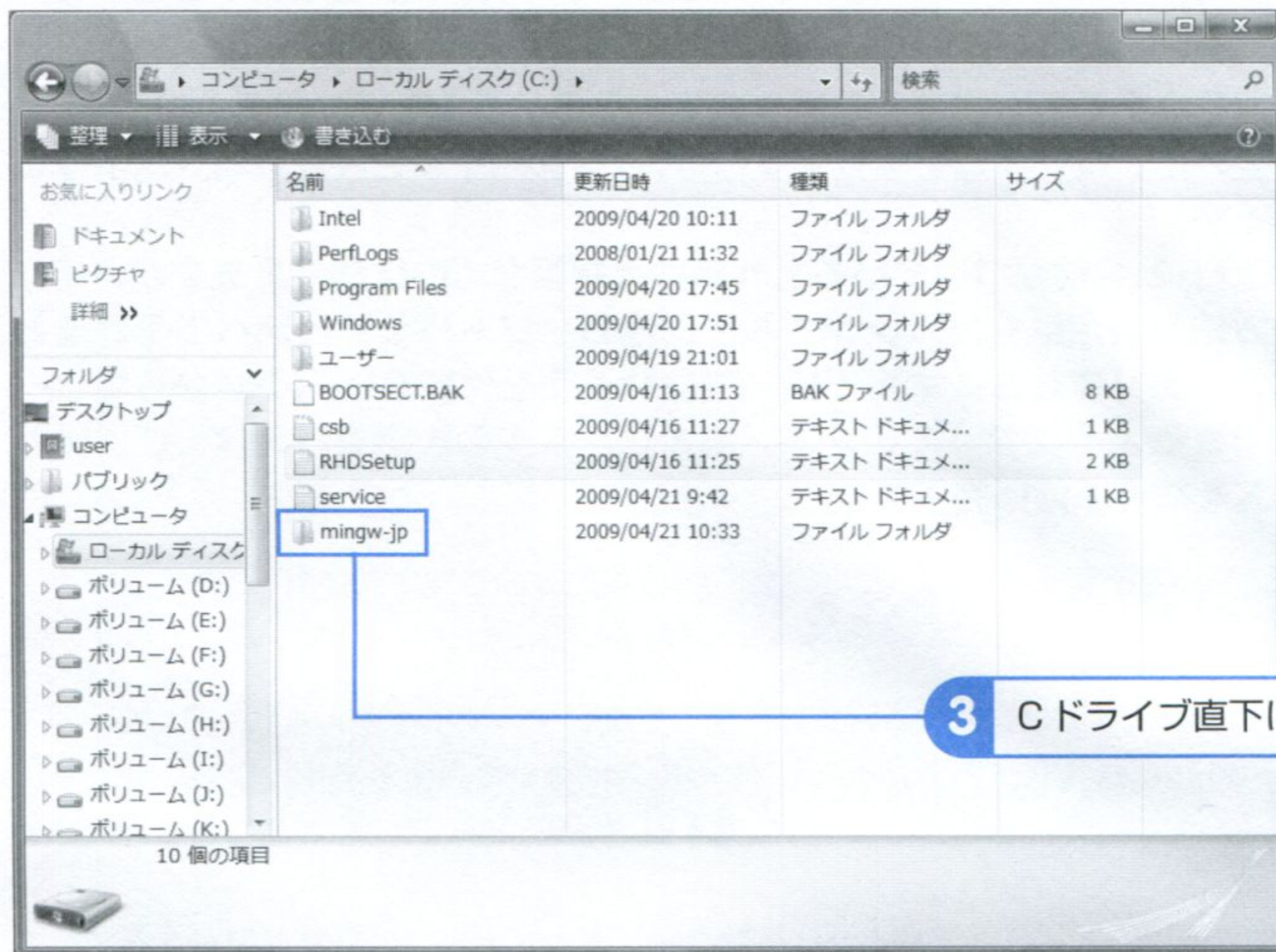
ヒント

*10：「スタート」－「すべてのプログラム」－「アクセサリ」－「エクスプローラ」でもエクスプローラを実行できます。

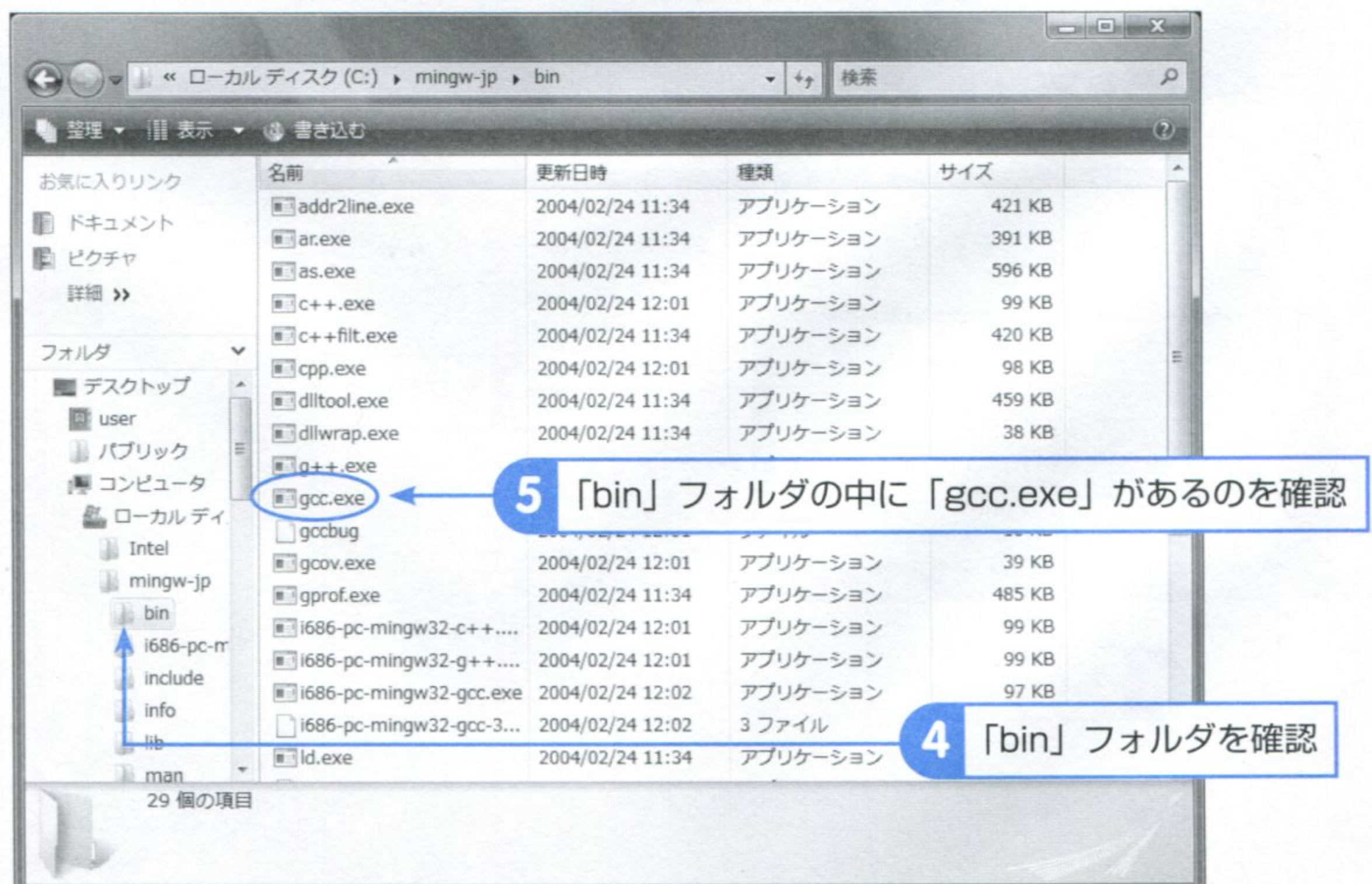
ヒント

*11：他の場所にコピーしてもかまいません。Cドライブ以外のドライブでも同じです。ただし、このコピーした場所は、あとで設定する「環境変数」と連動していますので、注意してください。

CD-ROMの「mingw-jp」フォルダごと、Cドライブ直下にコピーします*11。

3 Cドライブ直下にコピー

コピーが終了したら、C:¥mingw-jpの下にbinという名前のフォルダが存在することと、さらにその中にgcc.exe（またはgcc）というファイルがあることを確認してください。

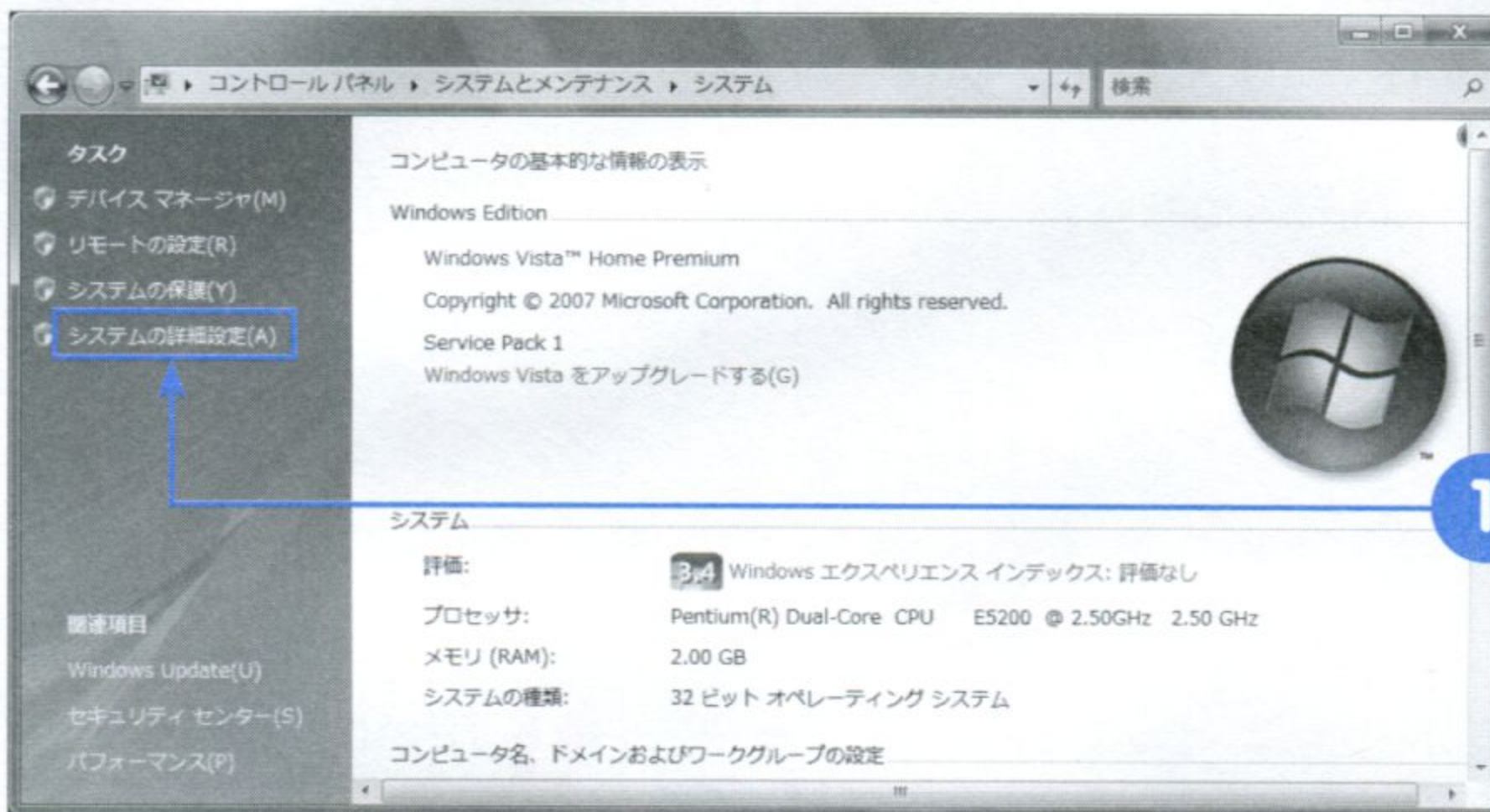
5 「bin」フォルダの中に「gcc.exe」があるのを確認

4 「bin」フォルダを確認

②環境変数を設定する

システムの環境変数にPATHを追加する設定を行うと、コマンドプロンプト画面でコンパイルを行うときに、自分がどこの階層にいても「gcc」というコマンド指定だけでコンパイルができます。

環境変数というのは、システム上で特定の名前の変数を作り、その変数にさまざまな値を設定したもので、システムやアプリケーションが使うための変数です。

このあたりの詳細は第1日で説明するので、今は深く考えず、設定だけ正確に行ってください。

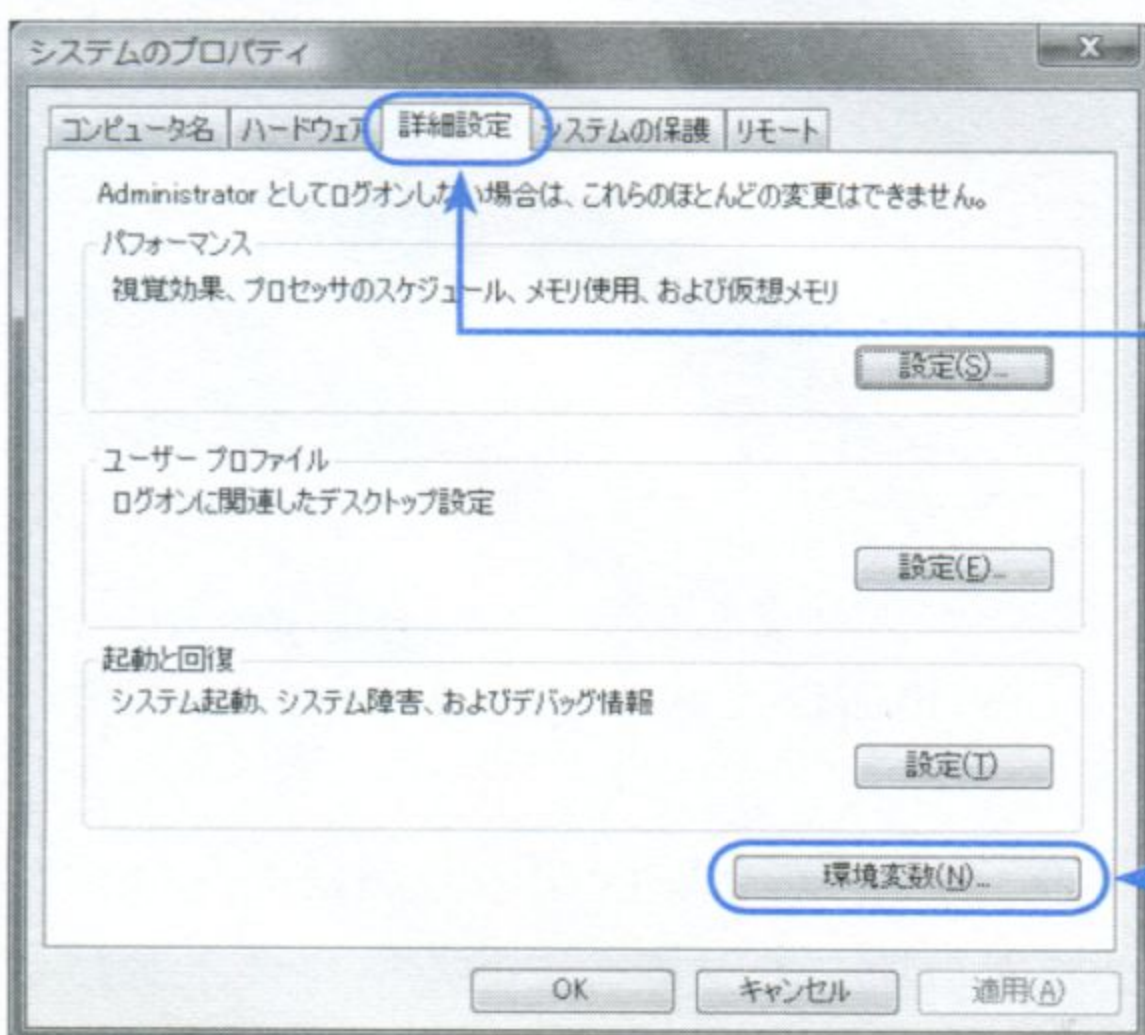
まず、「スタート」－「コントロールパネル」でコントロールパネルをひらき、「システムとメンテナンス」－「システム」でシステム画面まで移動します。このとき、管理者権限*12ではないアカウントでログインしている場合は、パスワードを入力する必要があります。タスクの「システムの詳細設定」をクリックすると、「システムのプロパティ」画面が表示されます*13。

ヒント

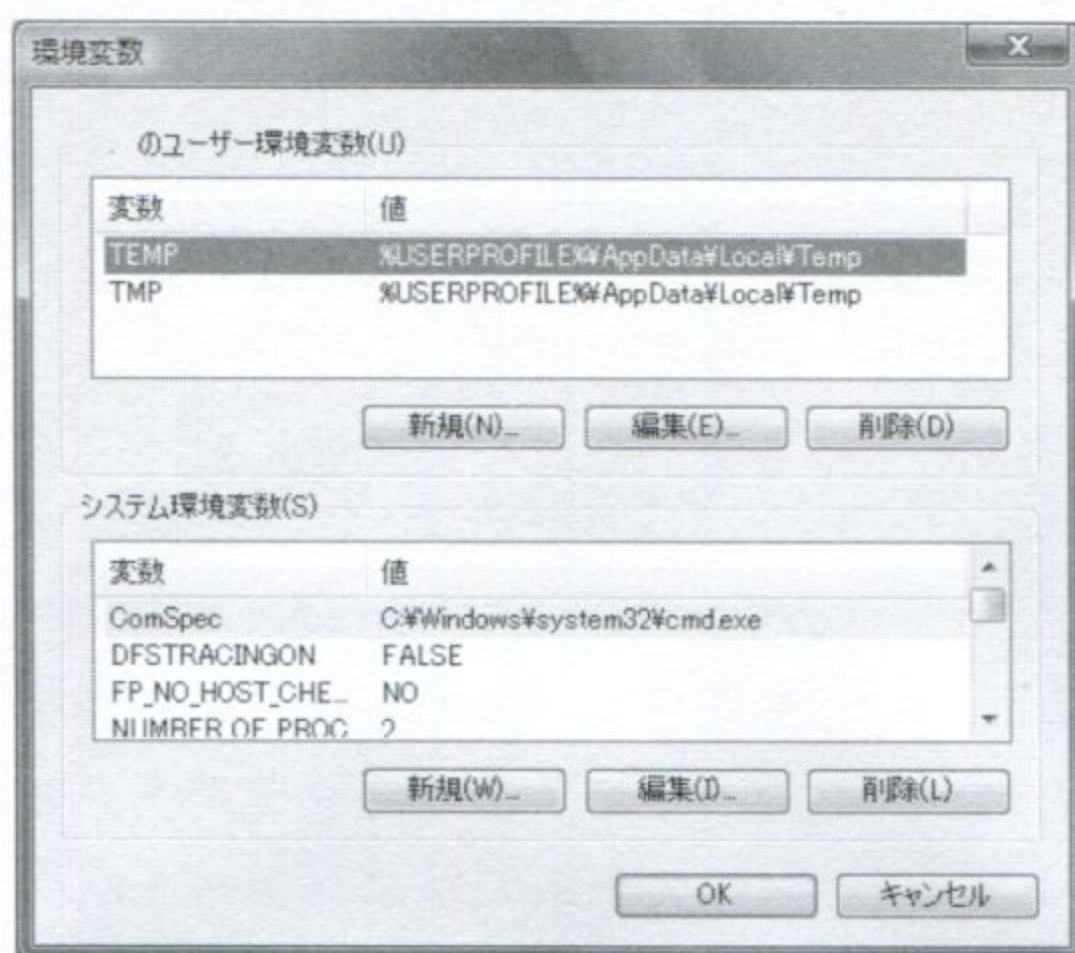
*12：Windows XPの場合は、あらかじめ管理者権限のあるアカウントで作業を行ってください。

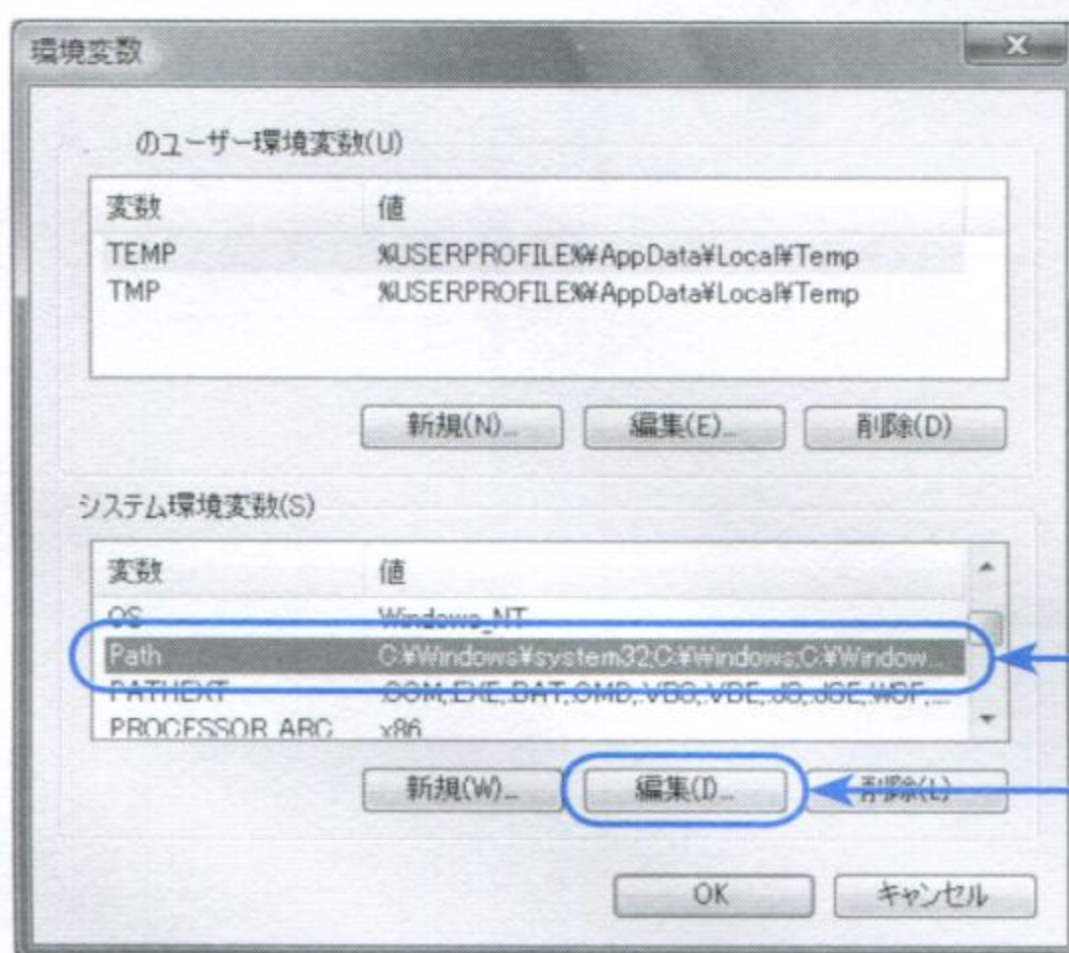
ヒント

*13：Windows XPでは「スタート」－「コントロールパネル」でコントロールパネルをひらき、「パフォーマンスとメンテナンス」－「システム」で「システムのプロパティ」画面を表示できます。

1 ここをクリック

「システムのプロパティ」画面の「詳細設定」タブで、下部の「環境変数」ボタンをクリックし、「環境変数」画面を表示してください。

2 ここをクリック

3 ここをクリック

③システム環境変数「Path」を設定する

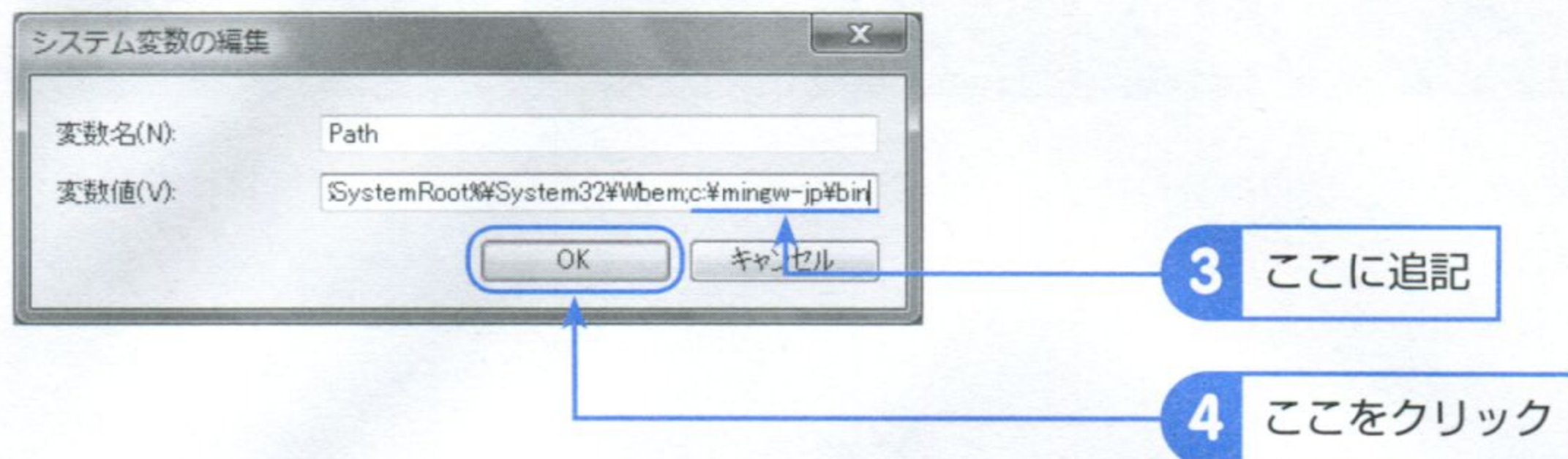
「システム環境変数」のPathというシステム環境変数名を選択し、「編集」ボタンをクリックします。PATHでもpathでも同じです。ここでは大文字小文字を区別していません。

すでに入っている変数値の最後に、

```
;c:¥mingw-jp¥bin
```

と続けて書きます。最初のセミコロン（;）を忘れずに入れましょう。「c:¥mingw-jp」は、先ほど、Cコンパイラをコピーした場所です。他の場所にコピーしたならば、その値に「¥bin」をつけて入力します。

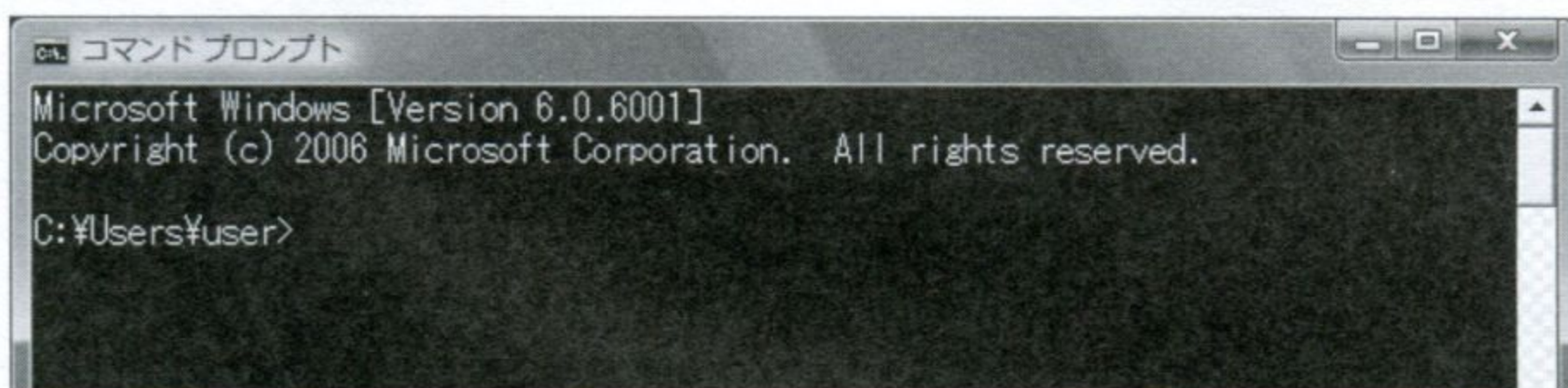
環境変数Pathの値は複数存在してもかまわないので、各値を区切るためにセミコロンを使います。もし、間違ってすでに設定してある変数値を消してしまった場合は、あわて

ずに「キャンセル」ボタンをクリックして、もう一度やり直しましょう。
　追加を行ったら「OK」ボタンをクリックします。

システム変数の編集
変数名(N): Path
変数値(V): ;SystemRoot%¥System32¥Wbem;c:¥mingw-jp¥bin
OK　キャンセル
3 ここに追記
4 ここをクリック

「環境変数」画面に戻るので、「OK」ボタンをクリックすると設定が適用されます。さらに「システムのプロパティ」画面でも「OK」ボタンをクリックして、終了してください。

Cコンパイラの確認を行う

　コンパイラのPathの設定が正しく行われているか確認します。
「スタート」－「すべてのプログラム」－「アクセサリ」－「コマンドプロンプト」で、コマンドプロンプトを起動します。

```
コマンド プロンプト
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>
```

「C:¥Users¥user>」といった表示の、最後の「user」部分はログインしたアカウント名になっているはずです。それぞれ違うはずなので、この部分は気にしないでください[*14]。

　キーボードから、単に「gcc」と入力して、[Enter] キー[*15]を押します。すると、

```
C:¥Users¥user>gcc ←「gcc」と入力して [Enter] キーを押す
gcc: no input files
```

こんな表示が出てきました。「gcc」と入力することで、「C:¥mingw-jp¥bin¥gcc.exe」というC言語のコンパイルプログラムを実行したことになります。gccプログラムは、本来、コンパイルするファイル名を引数[*16]（ひきすう）に指定して実行します。引数をつけずに実行すると、「gcc: no input files」とコマンドプロンプトに表示されます。

ヒント
＊14：コマンドプロンプトについての詳細はのちほど説明します。環境によっては「C:¥Users¥user>」の部分は、「C:¥Documents and Settings¥user>」や「C:¥Windows>」などになっている場合もあります。また、最初の「C:」が「D:」など他のアルファベットになっていることもあります。

ヒント
＊15：[Enter] キーとは、「Enter」と書かれたキーです。リターンキーとも呼びます。

ヒント
＊16：引数（ひきすう）とは、実行するプログラム名の最後に、ひとつ以上のスペースを入れ、そのあとに続けて書く文字列です。指定した引数の値は実行するプログラムに渡されます。gccでは通常、「gcc test.c」と、コンパイルを行いたいファイル名を引数に指定します。

gccを実行したあとに、

```
'gcc' は、内部コマンドまたは外部コマンド、
操作可能なプログラムまたはバッチ ファイルとして認識されていません。
```

と表示されたら、環境変数の値がうまく設定できなかったか、Cコンパイラ自体がおかしい可能性があります。

まずは環境変数の値を確認しましょう。

その他に準備すること

この他、プログラムを作りやすくするために必要な、Windowsの環境設定を紹介しておきます。

①ファイルの拡張子を表示する

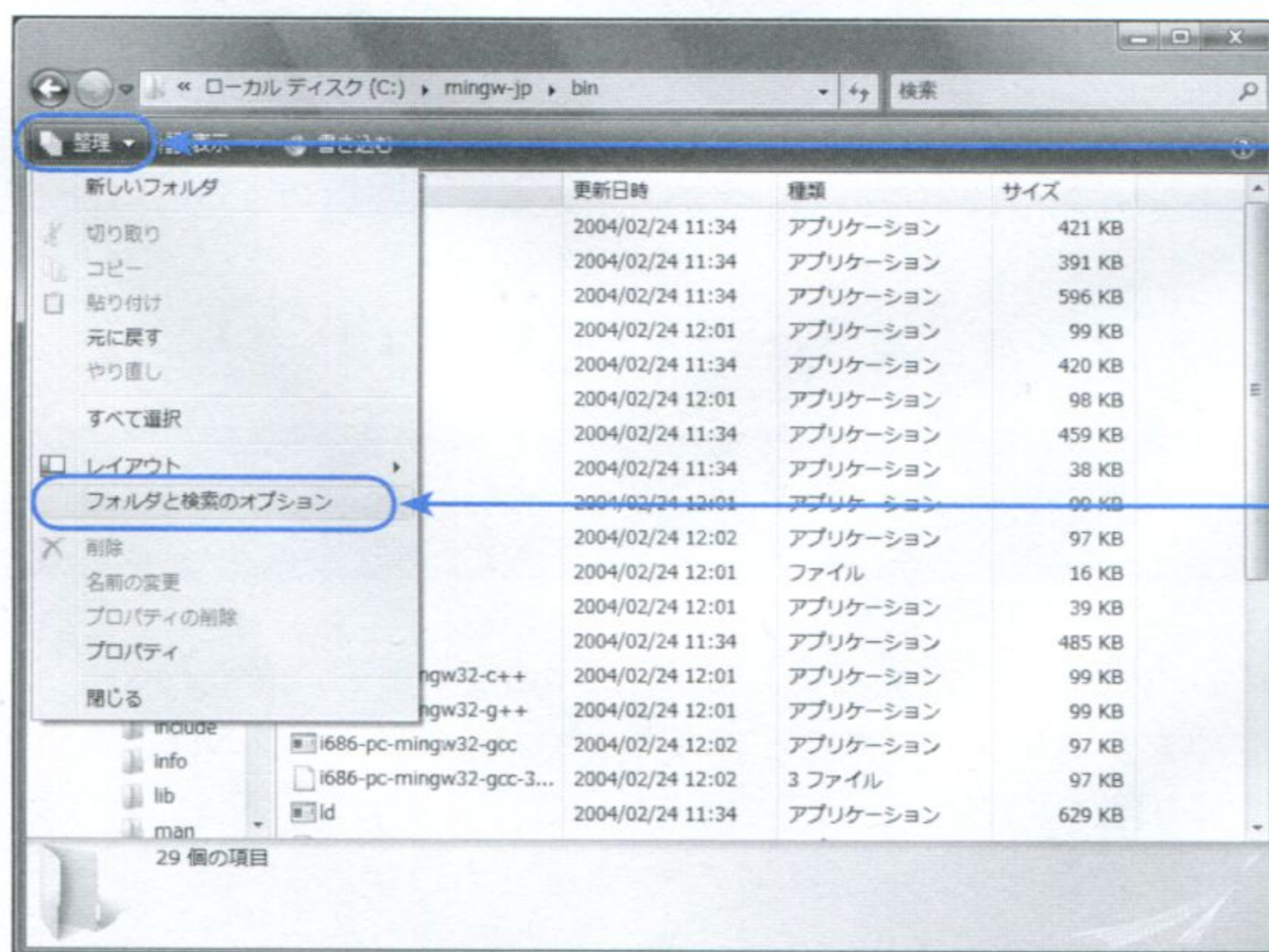
通常、エクスプローラでファイルやフォルダの一覧を確認すると、ファイルの拡張子が見えなくなっています。本書でこれから作成するプログラムファイルのファイル名とその実行ファイルのファイル名は拡張子が異なるだけなので、このままでは区別がつきません。

ファイルの拡張子が見えるように変更しておきましょう。

エクスプローラで、Cコンパイラをインストールしたフォルダ「C:¥mingw-jp」の、さらにその中にある「bin」フォルダを開きます。

コマンドバーの「整理」－「フォルダと検索のオプション」で、「フォルダオプション」画面を開きます[*17]。

ヒント

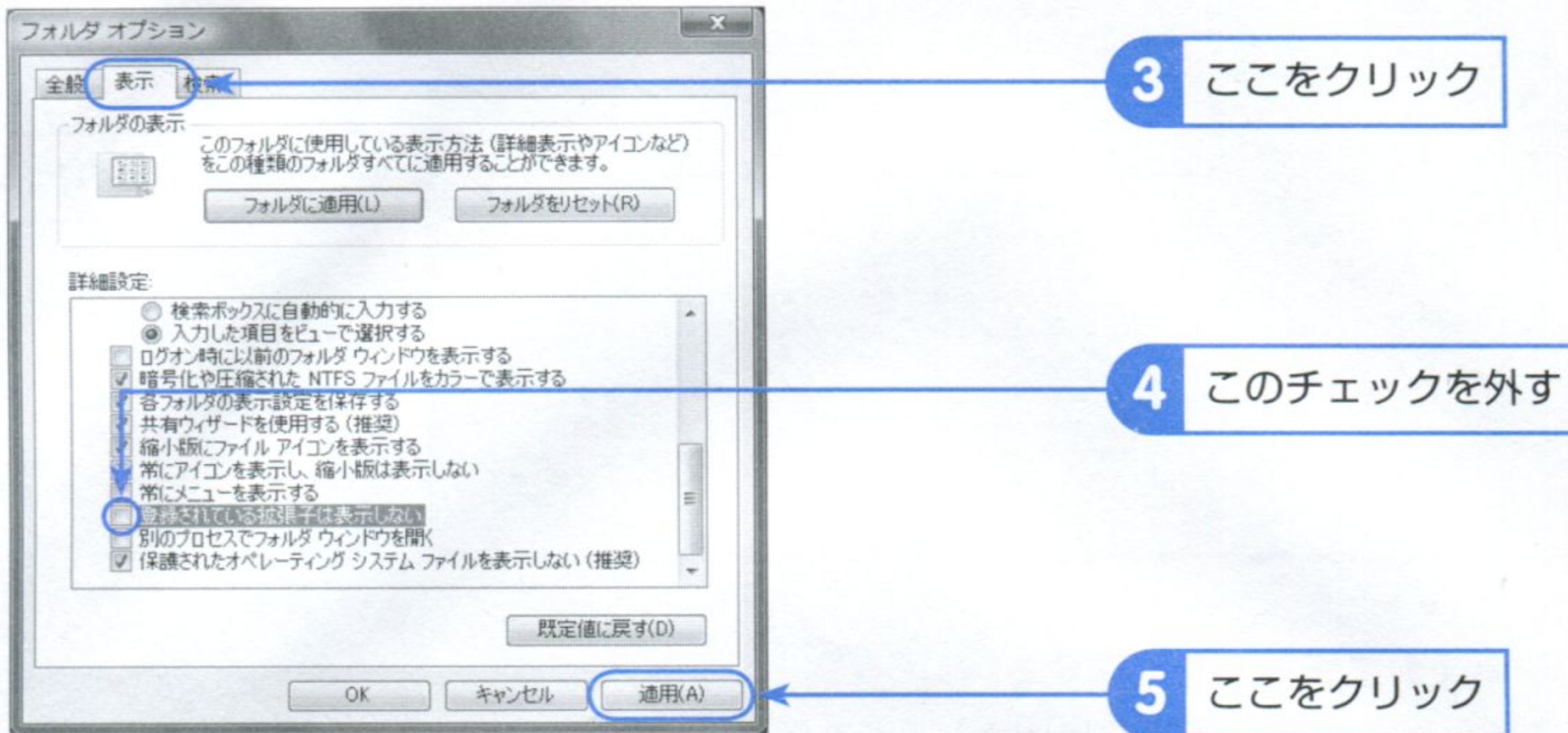
*17：Windows XPではエクスプローラの「ツール」－「フォルダオプション」の「表示」タブで設定します。

1 ここをクリック

2 ここをクリック

「表示」タブの「詳細設定一覧」にある、「登録されている拡張子は表示しない」のチェックをはずしてから、「適用」ボタンをクリックします。

3 ここをクリック

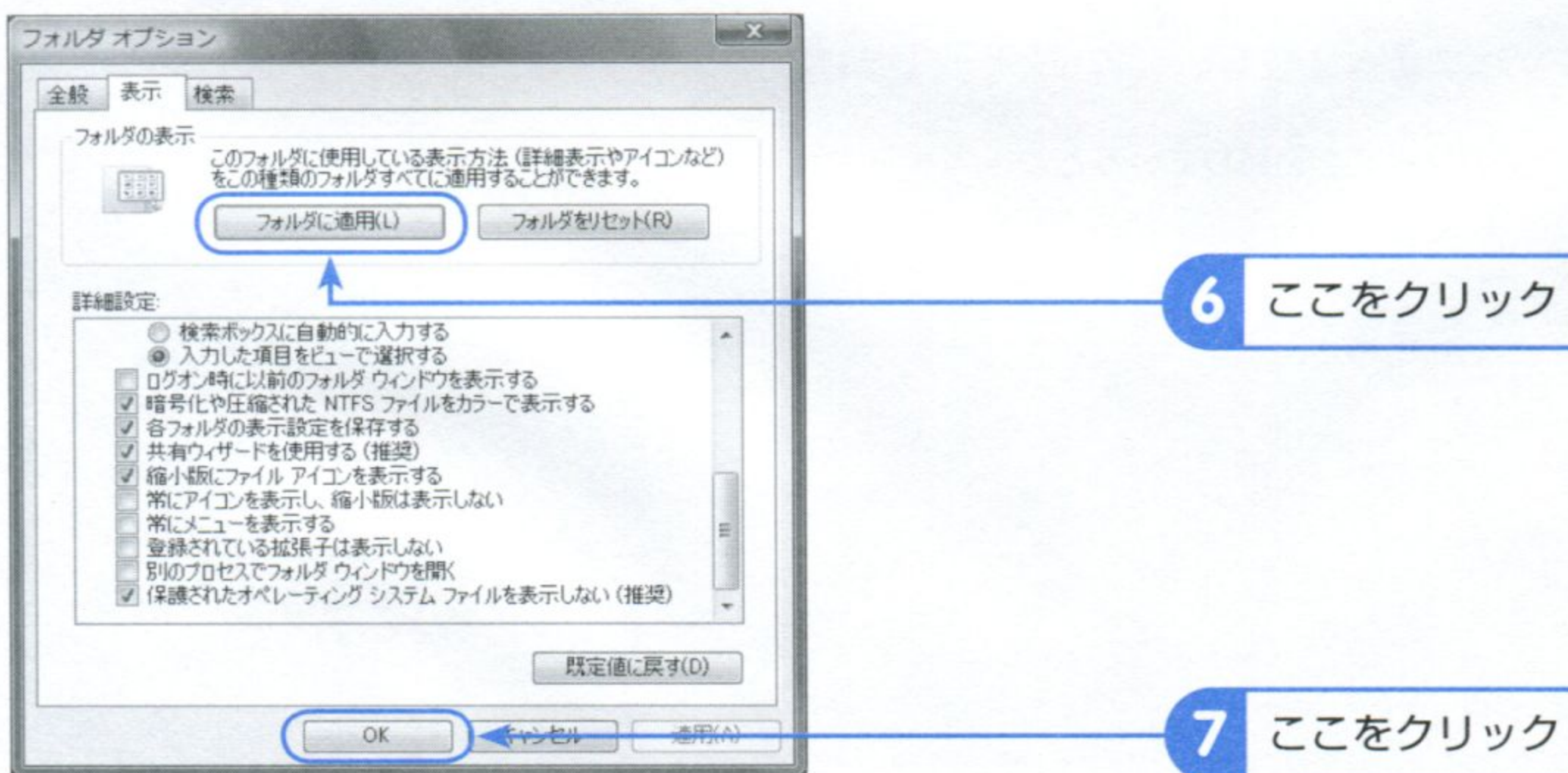
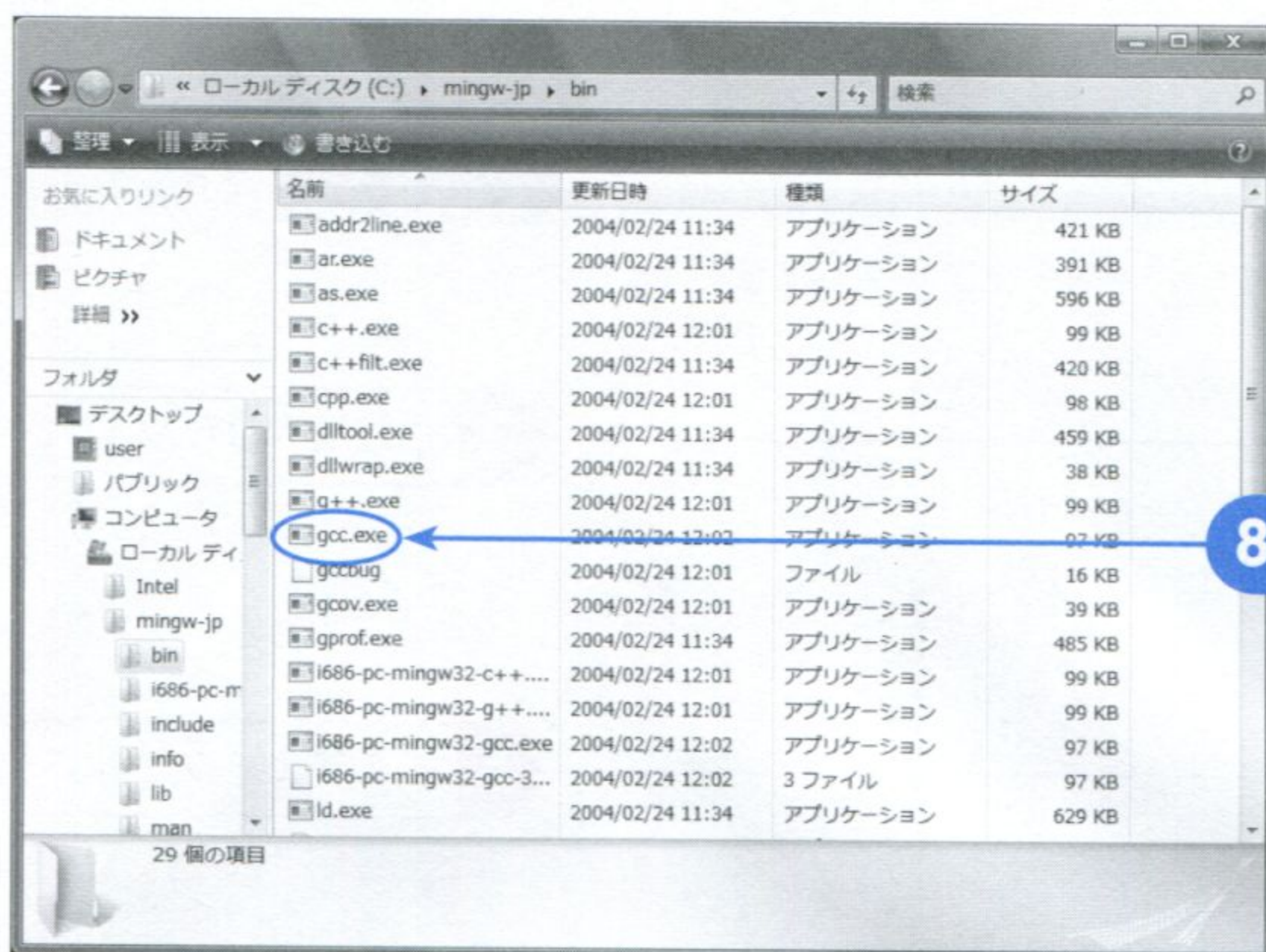
4 このチェックを外す

5 ここをクリック

次に、すぐ上の「フォルダの表示」にある「フォルダに適用」ボタン[*18]をクリックして、メッセージが出たら、「はい」を選択します。最後に「OK」ボタンをクリックして終了します。

ヒント

*18：Windows XPでは「全てのフォルダに適用」ボタンをクリックして、メッセージが出たら、「はい」を選択します。

6 ここをクリック

7 ここをクリック

「C:¥mingw-jp¥bin」や他のフォルダを見て、拡張子が見えるようになっているか確認してください。

②テキストエディタを用意する

C言語のプログラムを書くにはテキストエディタを使用します。

Windows付属のテキストエディタである「メモ帳」を使ってもよいのですが、他に使い勝手のよいものがあれば、それを使用してください。インターネット上ではフリーのソフトウェアも数多く紹介されています。

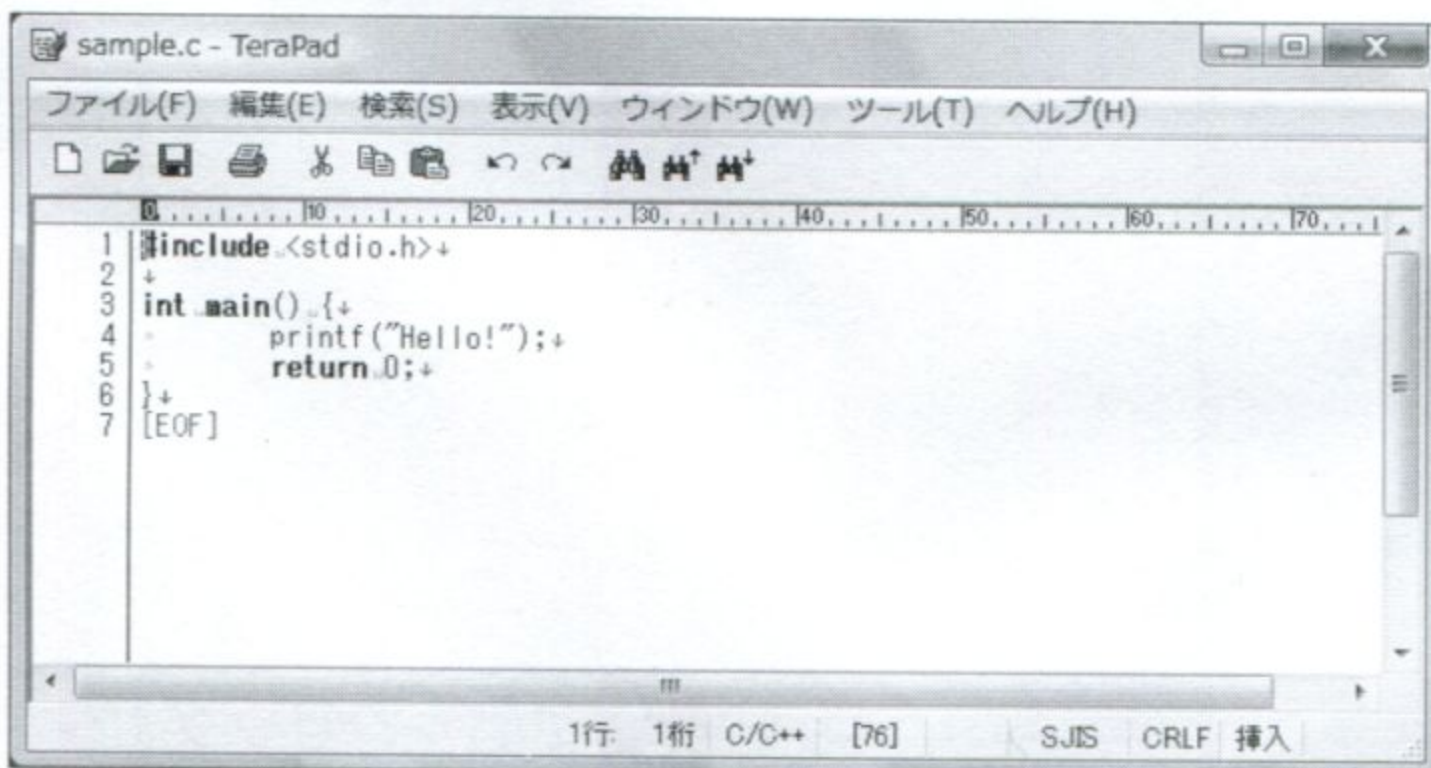
テキストエディタの中には、プログラムを書くときにC言語の予約語やコメント部分を別の色や太さで表示してくれるものもあります*19。自分で使いやすく、かつ便利なものを用意してください。

ヒント

*19：テキストエディタ上の文字の色や太さはあくまで表示上のものです。テキストエディタで作ったテキストファイルは、レイアウト情報を持たない文字データです。

本書ではフリーソフトTeraPadを推奨します。

●TeraPadでソースを記述しているところ

```
#include <stdio.h>

int main() {
	printf("Hello!");
	return 0;
}
[EOF]
```

まとめ

ここまでの設定で、C言語のプログラムを書くための準備は整いました。

明日からは実際にプログラムを書きながら、本格的にC言語の勉強を進めましょう。

第 1 日

はじめての C言語プログラムを作ろう

1時限目　一番簡単なプログラムを作って動かそう

2時限目　相性占いプログラムを作ろう

3時限目　コマンドプロンプトを使ってみよう

ここからがC言語学習の本当の第一歩です。
第0日のオリエンテーションでC言語のプログラムを書く準備は整ったと思います。
まだ準備をしていない人は、第0日に戻って環境を整えてください。
本章から実際にC言語のプログラムを書いて、実行する学習に入ります。まずはとても簡単なプログラムを作って、プログラムの書き方の基本を理解し、その次に少しだけ複雑なプログラムを作って、プログラムを書くことに慣れましょう。

第1日 1時限目

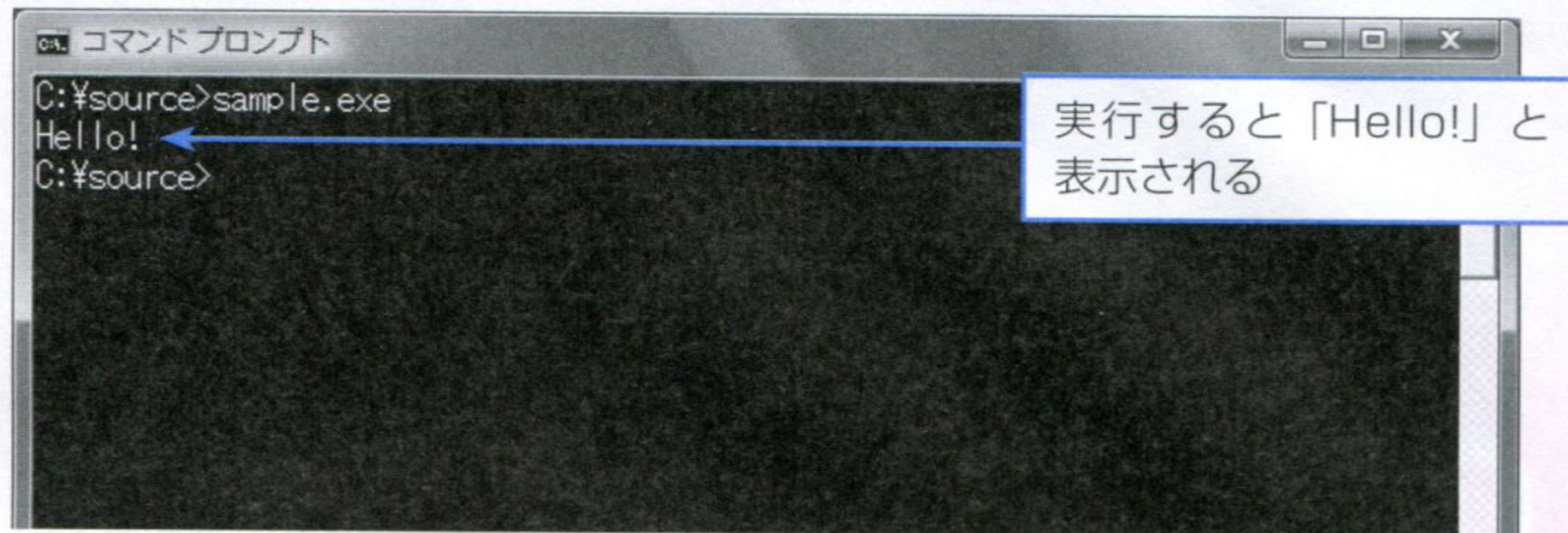
【はじめてのC言語プログラムを作ろう❶】

一番簡単なプログラムを作って動かそう

お待たせしました、この時間からやっと実践に入ります。最初から複雑なプログラムを書いても、結局実行できずにストレスを溜めただけ……なんてことにならないように、まずはごく簡単なプログラムを書いて、実行します。

今回作成する例題

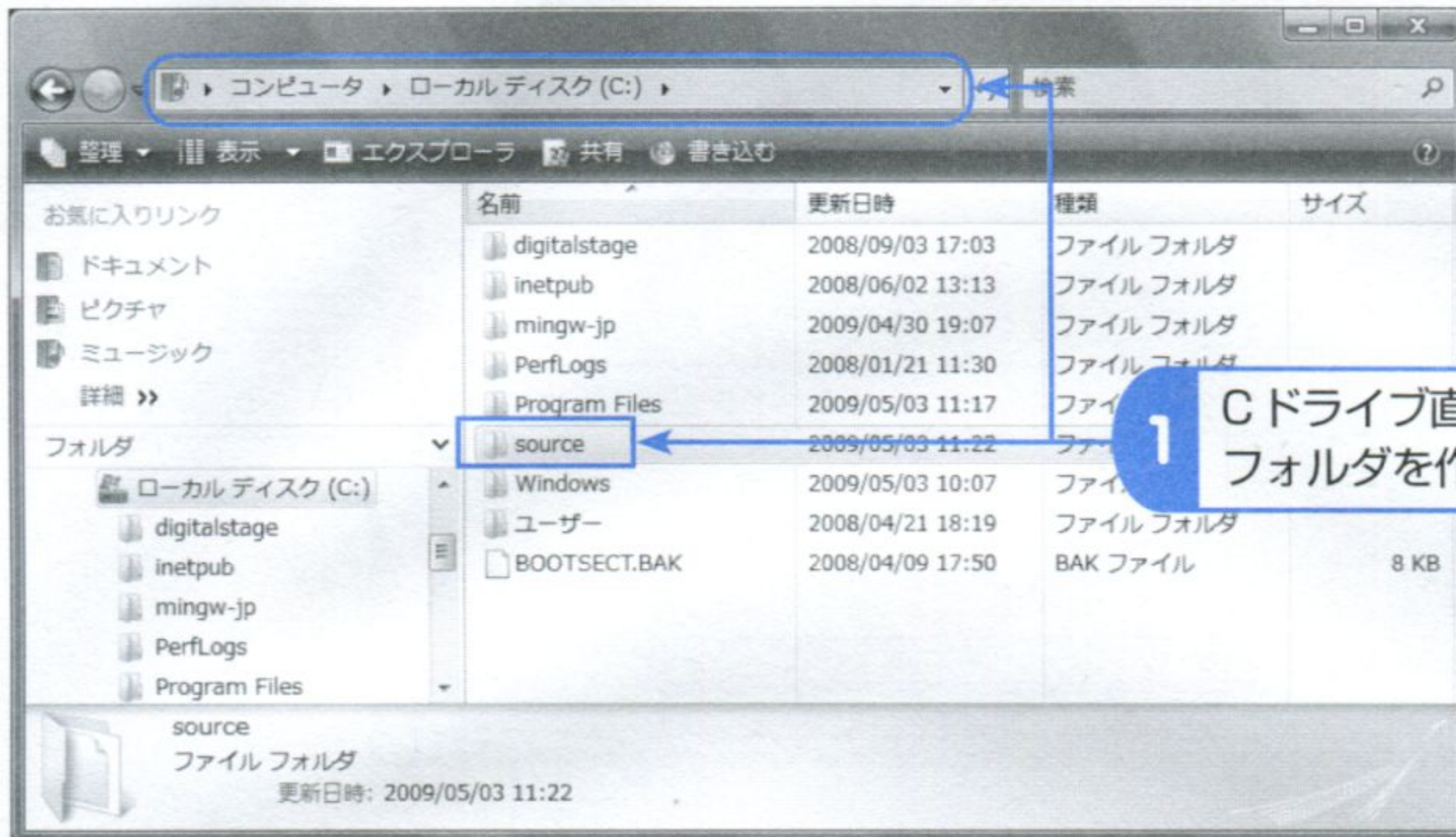
```
コマンド プロンプト
C:¥source>sample.exe
Hello!
C:¥source>
```

実行すると「Hello!」と表示される

サンプルファイルはこちら 10days_c ▶ day01-01 ▶ sample.c

●このレッスンのねらい

C言語で書く、一番基本的なプログラムを実際に作ってみます。
プログラムの書き方、保存の仕方、コンパイルの仕方など、実践的な技術を学びながら、C言語の文法の基本をマスターします。
この時限で勉強する「プログラムを書く」→「コンパイルする」→「実行する」という流れは、この先どんなにプログラムが複雑になってもかわらない手順です。
この時限以降では特に説明しないので、ここで慣れてしまいましょう。

プログラムを作成する

1 Cドライブの下に、「source」という名前のフォルダを新規作成する

2 テキストエディタ[*1]で新規文書を作成し、次のコードを入力する[*2]

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello!");
5      return 0;
6  }
```

最後の行も改行する

ヒント

*1：ご自分で用意したテキストエディタを使ってください。Windows付属のメモ帳でもかまいません。

ヒント

*2：行番号は、プログラム中には書かないでください。

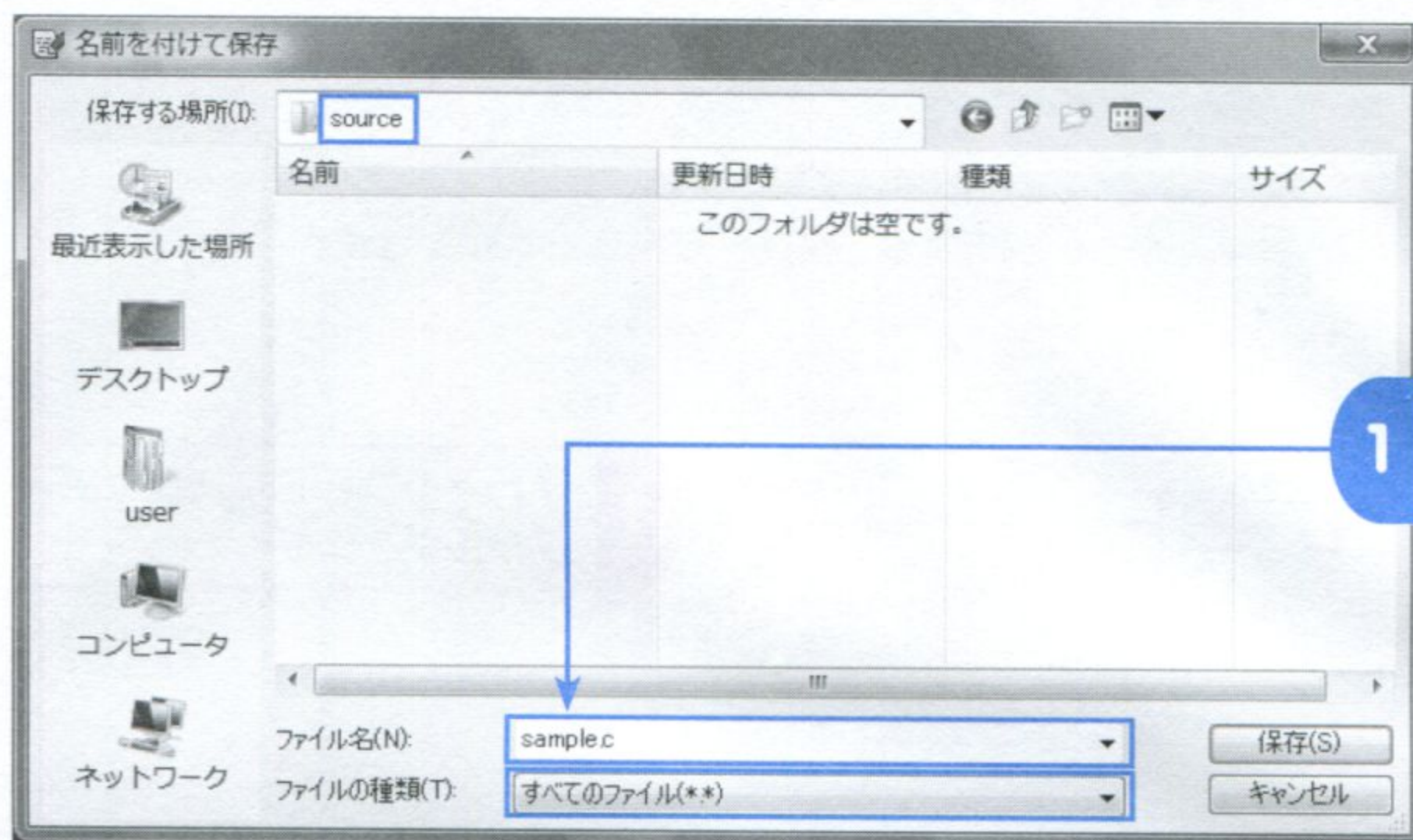
3 入力できたら「sample.c」という名前[*3]で、手順①で作成した「C:¥source」フォルダ下に保存する[*4]。保存するときのファイルの種類は、「すべてのファイル」を選択しておく[*3]

名前を付けて保存

保存する場所(I): source

名前	更新日時	種類	サイズ
	このフォルダは空です。		

最近表示した場所
デスクトップ
user
コンピュータ
ネットワーク

ファイル名(N): sample.c
ファイルの種類(T): すべてのファイル(*.*)
保存(S)
キャンセル

1 「sample.c」という名前で保存

4 「スタート」－「すべてのプログラム」－「アクセサリ」の「コマンドプロンプト」を起動し、cdコマンドを入力して、「C:¥source」フォルダに移動する

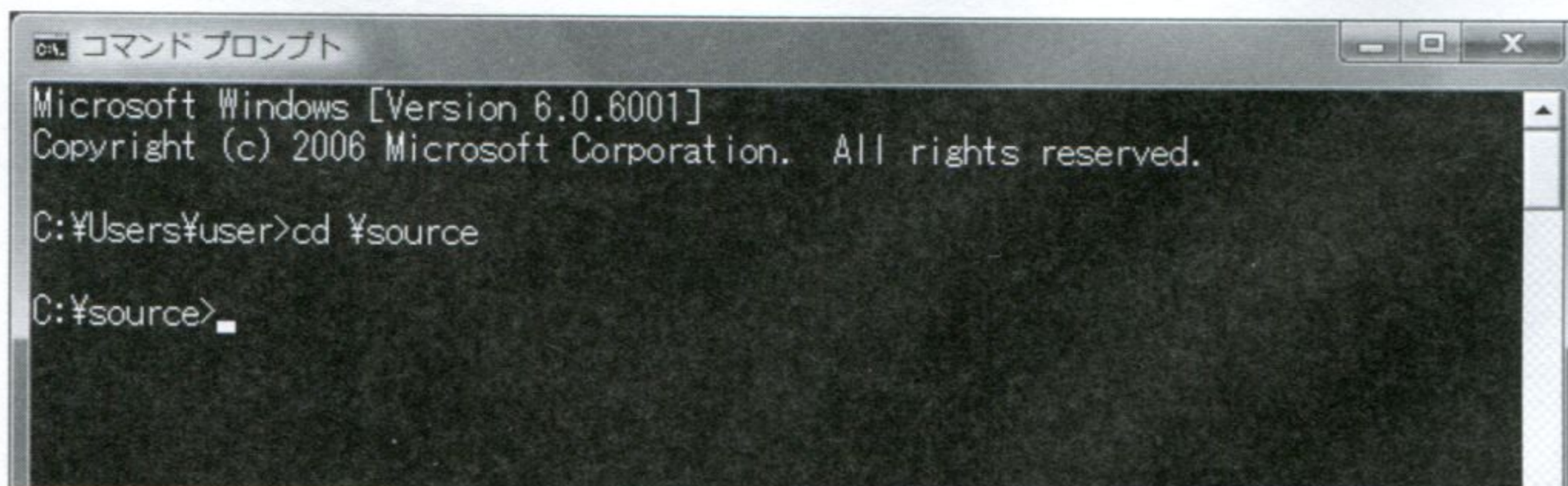
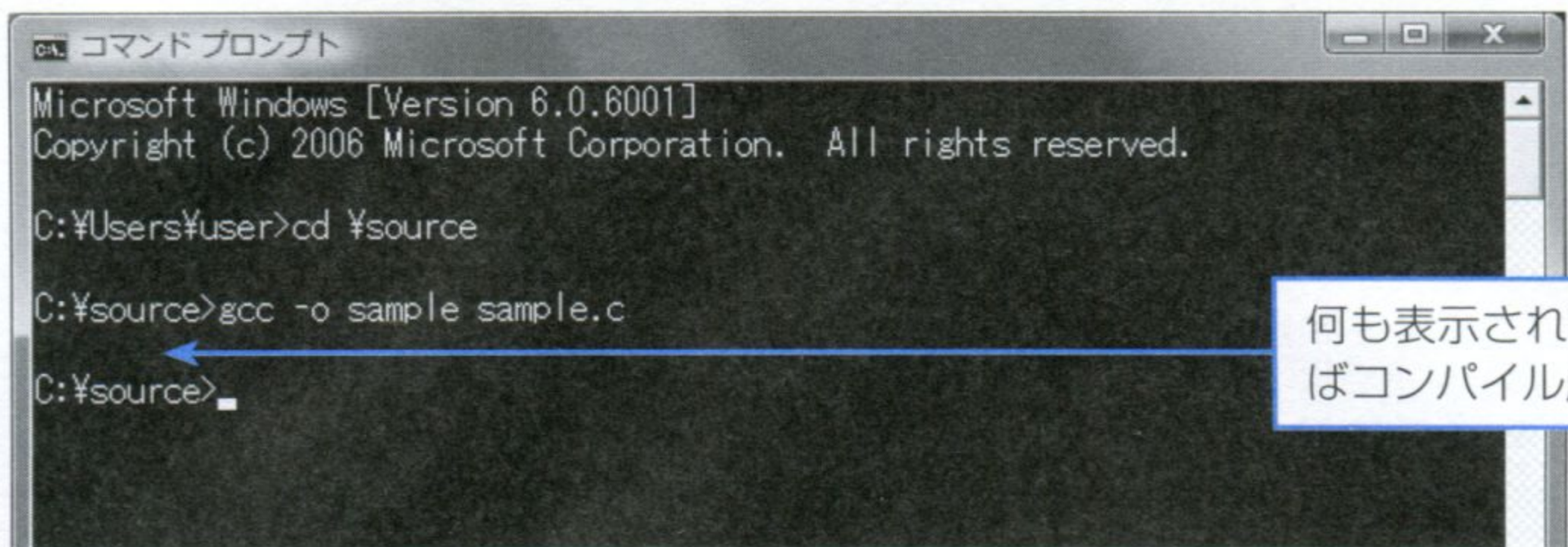
```
C:¥Users¥user>cd ¥source
```

←「cd ¥source」のみ入力して［Enter］キーを押す[*5,*6]

```
C:¥source>
```

←「C:¥source>」に変更になった

コマンド プロンプト

```
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>cd ¥source

C:¥source>_
```

5 次のように入力して、sample.cをコンパイルする[*7]

```
C:¥source>gcc -o sample sample.c
```

←入力するのは「gcc -o sample sample.c」のみ

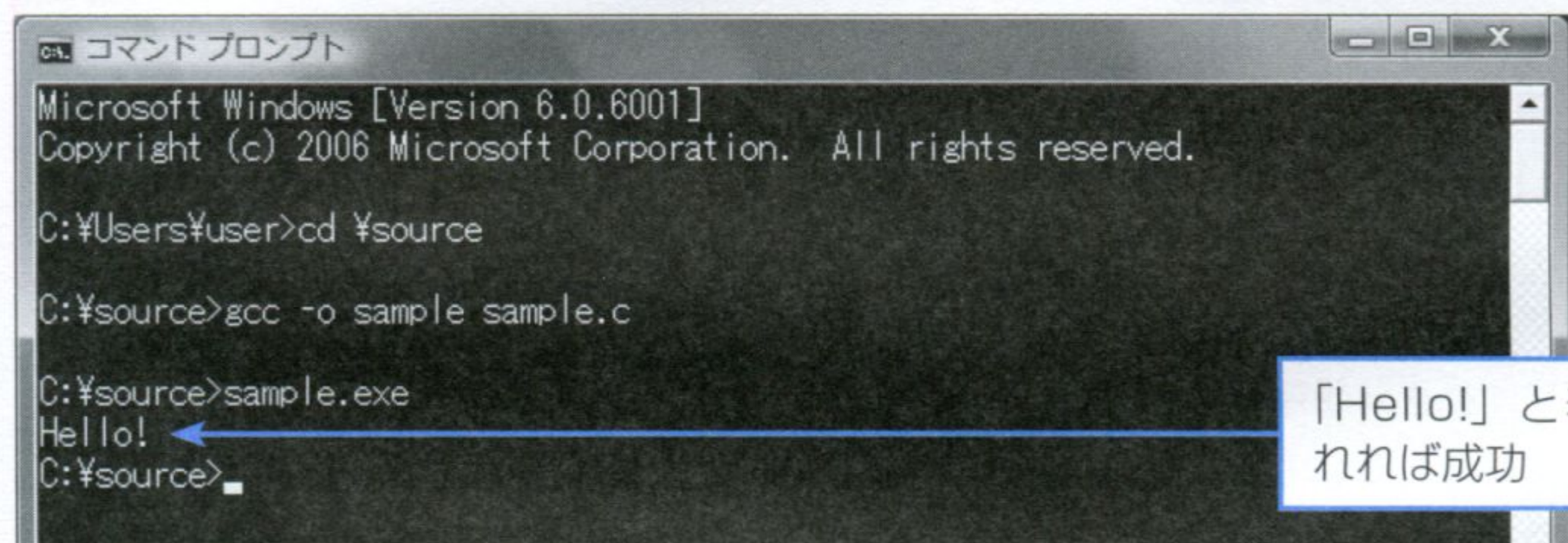
ヒント
*3：拡張子に注意して保存しましょう。ファイルの種類をうっかり「テキストファイル」で保存すると、拡張子に.txtがついて、「sample.c.txt」なんて名前になってしまいます。

ヒント
*4：保存するフォルダは、他のところでも構いませんが、その場合は、以下の「C:¥source」を保存したフォルダに読み替えて実行してください。

ヒント
*5：コマンドプロンプトの表示が「C:¥Users¥user>」ではなく「C:¥Documents and Settings¥user>」になっている環境もありますが、「cd ¥source」だけ入力しましょう。

ヒント
*6：sample.cを保存した場所が「C:¥source」で、コマンドプロンプトを起動したときの表示が「C:¥」以外ではじまる場合は、まずは「c:」と入力して［Enter］キーを押してください。それから「cd ¥source」を入力します。

ヒント
*7：コンパイルが成功しない場合、P.38以降の説明を読んで原因を探ってください。

```
コマンド プロンプト
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>cd ¥source

C:¥source>gcc -o sample sample.c

C:¥source>
```

何も表示されなければコンパイル成功

6 次のように入力して、プログラムを実行する

```
C:¥source>sample.exe
```

入力するのは「sample.exe」のみ

```
Hello!
```

```
コマンド プロンプト
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>cd ¥source

C:¥source>gcc -o sample sample.c

C:¥source>sample.exe
Hello!
C:¥source>
```

「Hello!」と表示されれば成功

解説

1 プログラムの作成と実行の手順

C言語で書いたプログラムを実行するためには、以下の手順でプログラムソースファイルから実行ファイルを作り出す必要があります。

●C言語で書いたプログラムの実行手順

プログラムソースファイルsample.c

```
#include <stdio.h>

int main() {
    printf("Hello!");
    return 0;
}
```

テキストエディタで作成・保存
(C:¥source¥sample.c)

↓

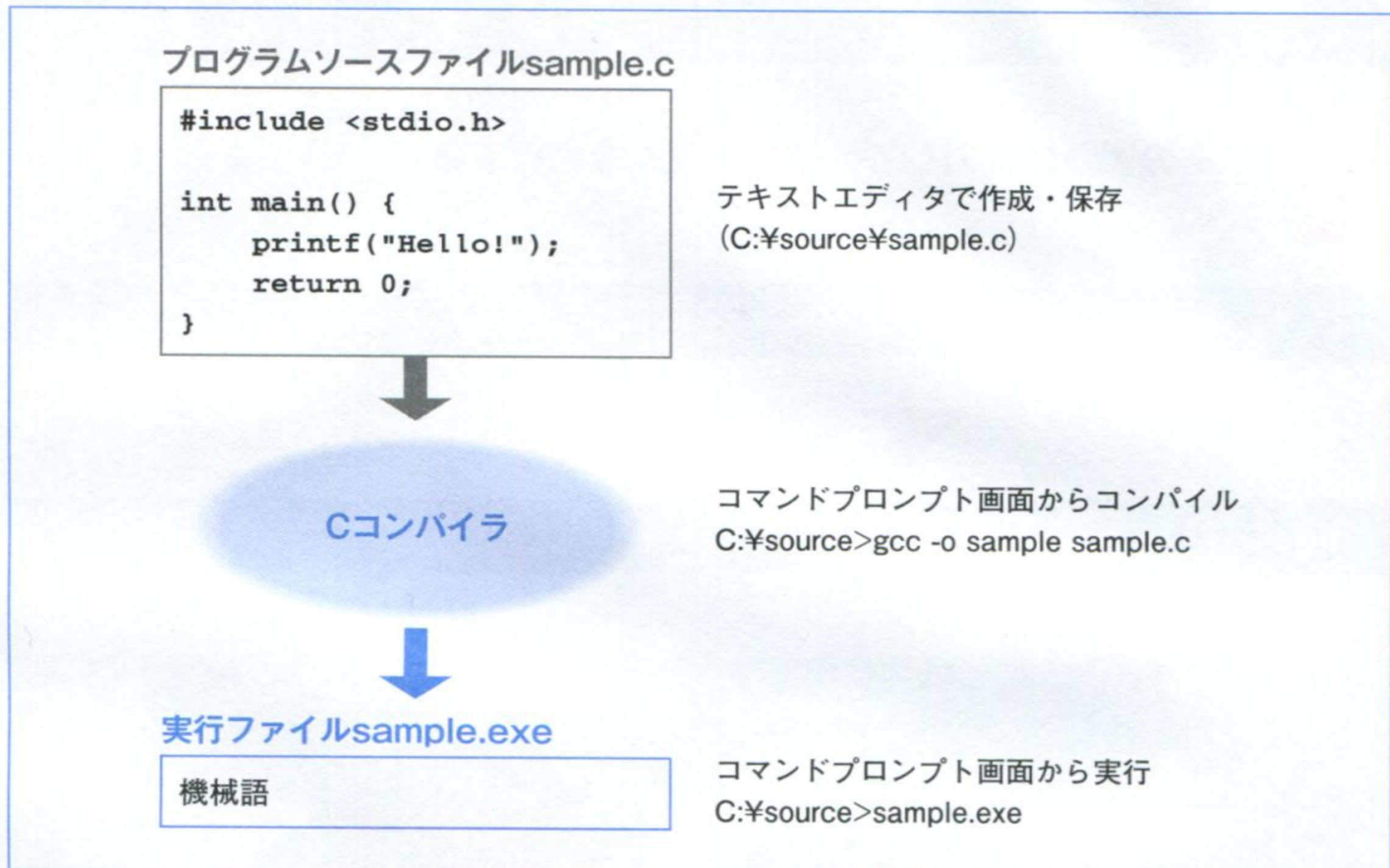
Cコンパイラ

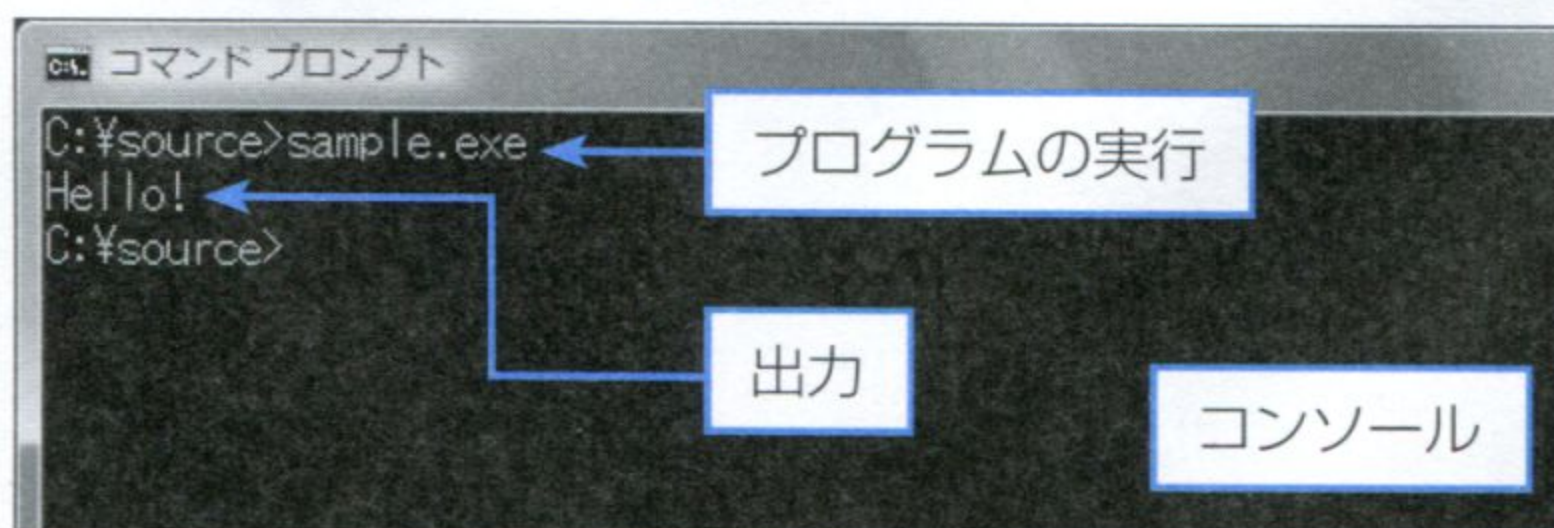
コマンドプロンプト画面からコンパイル
C:¥source>gcc -o sample sample.c

↓

実行ファイルsample.exe

機械語

コマンドプロンプト画面から実行
C:¥source>sample.exe

最初にプログラムソースファイルを作成し、次にプログラムソースファイルをコンパイルします。このとき、リンクと呼ばれる作業も、Cコンパイラが同時に行ってくれます。最後に、できあがった実行ファイルを実行します。

今回作ったプログラムは、コンソール*8上に「Hello!」とだけ出力表示するプログラムです。プログラムがコンソール上に対して何かを書き出すことを、出力といいます。

ヒント
*8：コマンドプロンプト画面をコンソール画面ともいいます。「コンソール上」とは、コンソール画面の大部分を占める文字表示部分を指します。

●コンソール上に実行結果を表示

```
コマンド プロンプト
C:¥source>sample.exe
Hello!
C:¥source>
```

プログラムの実行
出力
コンソール

2 プログラムの文法

オリエンテーションでも説明しましたが、プログラムとはコンピュータに行ってほしい動作の手順を書いたものです。C言語でプログラムを書く場合は、C言語の文法に従って書きます。例えば、C言語のプログラムは必ず、

```
int main() {

}
```

という部分が必要になります。このmain()部分からプログラムは実行されます。

これはC言語の「きまり」です。プログラムは、このようなきまりごと（文法）に従って書く必要があります。

人間同士のコミュニケーションでは、「この部分がmainです」と書くのを間違えて「この部分がmainnです」と書いても、「あ、単語を書き間違えているのね」で済みますが、コンピュータの場合、そのような柔軟な思考はもっていません。

```
int mainn() {

}
```

と書いてあったら、「C言語のプログラムなのにmain()部分がないぞ！」とコンピュータに怒られてしまいます。コンピュータプログラムの文法は、正確に書かなければなりません。

このmain(){}部分は、単にmain（メイン）とかmain関数と呼ばれます。

「main()」のあとの左括弧「{」から右括弧「}」までがmainです。{}で囲まれた部分をブロックと呼びます。

3 ソースコードの説明

それでは、ソースコードを１行目からじっくりと説明していきます。

(1) 1行目

出力を行うprintf関数を利用するためのおまじないです。

```
#include <stdio.h>
```

と書いてあった場合、これは「stdio.hファイルをインクルードする」といいます。

詳しい説明は第10日にしますが、今は、「printf関数を使うためには、必ずプログラムの最初にこの１行を書かなければならない」ということだけ、おぼえておいてください。

関数とは特定の処理を行うもので、printf関数は文字列を出力する関数です。C言語で用意されているさまざまな関数を使うためには、他にもいろいろな「○○ファイルをインクルードする」必要があります。stdio.h以外のファイルをインクルードする必要が生じたら、そのつど説明します。どのプログラムでもたいていprintf関数は使うはずなので、C

言語のプログラムを書くときは、1行目に必ず「#include <stdio.h>」を書くことに慣れてしまいましょう。

(2) 2行目

プログラムの見た目をスッキリさせるために、1行分、空けます。

(3) 3行目〜6行目

C言語のプログラムはmain() { }の中からはじまります。main関数の中の各命令文をタブ（[Tab] キー）で揃えると、見やすいプログラムになります。[Tab] キーなどで字下げすることを、インデントといいます。

```
int main() {
        命令文;
        命令文;
        命令文;
}
```

プログラムはコンピュータへの命令手順書ですから、main関数の中の最初の文から順に実行することを考えて、プログラムを書くのが基本です。main関数の中の処理は、必ず{}で括ります。

main()の前についている「int」は、5行目の「return 0;」に対応しているものです。今はまだ深く考えずに書いてください。

(4) 4行目

main関数の中の最初の命令文です。ここからプログラムは実行されます。
printf関数を使って文字列「Hello!」を出力します。

(5) 5行目

mainブロックの最後です。プログラムが正常におわった証拠に、0を返します。mainブロックの最後には必ず「return 0;」をつけましょう。

とても短いプログラムなので、プログラムの中身と呼べるのは、4行目のみです。
残りの部分は、他にどんな内容のプログラムを書いても、同様に書く必要があります（最初の「#include <stdio.h>」は、printf関数を使用しないのであれば、必要ありません）。
C言語のプログラムは、すべて次のひな型をもとに書きましょう。

●C言語プログラムのひな形

```
/* インクルード文を書く */

int main() {
        /* 処理を書く */
        return 0;
}
```

4 関数とは？

main関数の中に実行したい命令文を順に書いていくと、プログラムができあがります。例えば、

```
#include <stdio.h>

int main() {
    printf("Woo!");
    return 0;
}
```

と書いたプログラムを実行すると、文字列「Woo!」が表示されます。printfは出力を行う命令文です。他にも目的にあわせた命令文がそれぞれ用意されていて、それらを使ってプログラムを書きます。このそれぞれの命令文は、プログラム言語によって異なります。日本語や英語の違いでいうと、単語の違いに該当します。

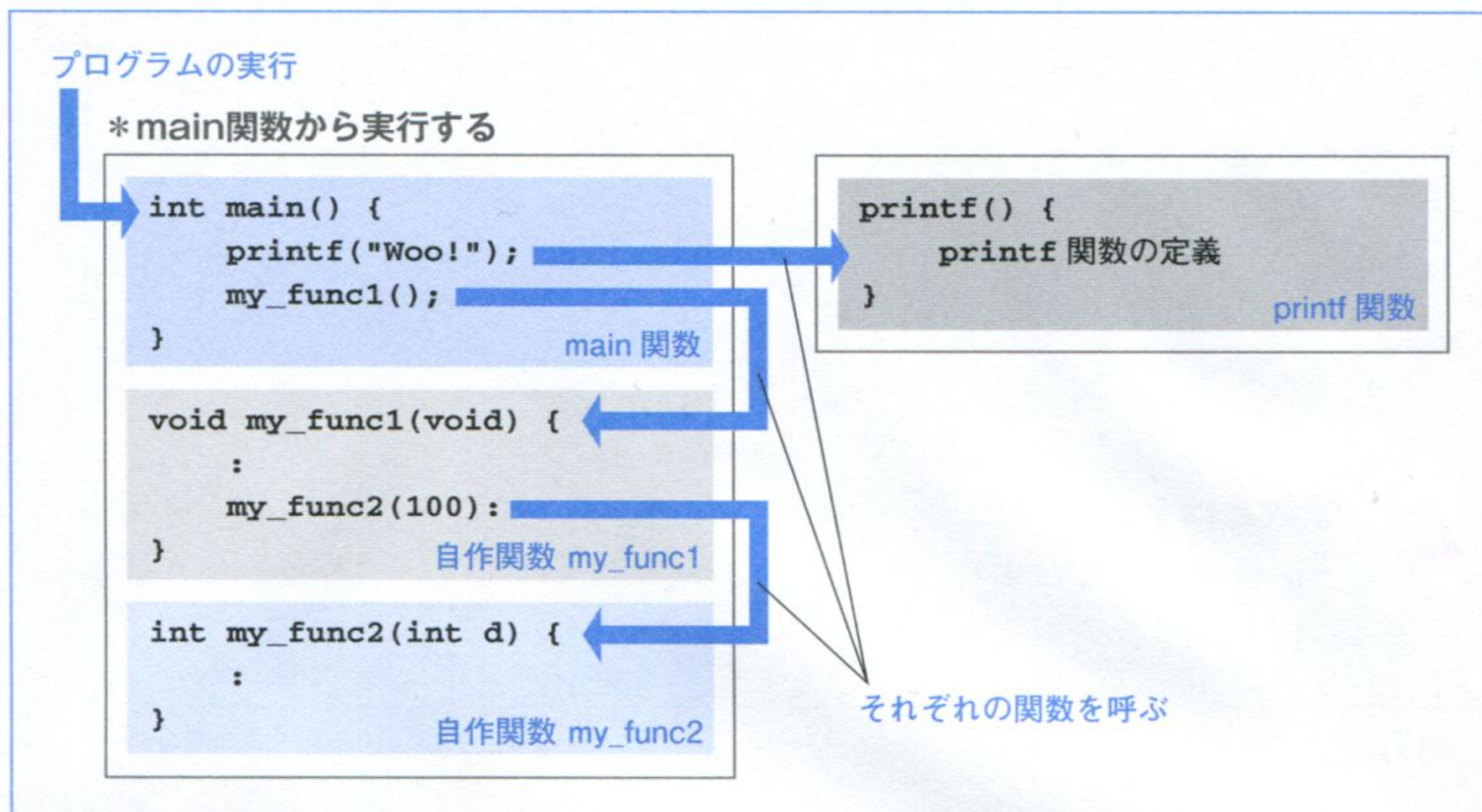
これらの用意されている命令文は、○○関数と呼びます。printf命令文はprintf関数といいましょう。

C言語では数多くの関数が用意されていますが、すべてを完璧におぼえておく必要はありません。本書でも、使用するものだけを少しずつ説明していきます。よく使う関数は自然とおぼえてしまうものですから、無理におぼえようとせずに気楽に学習しましょう。

C言語で使う関数はどこかに定義されていて、main関数の中から呼び出して使用します。また、mainも関数のひとつです。プログラムを実行すると、最初にmain関数が呼ばれるきまりになっています。

いくつかの処理をまとめて自分で関数を作ることもできます。作った関数は、main関数やまた別の関数から呼び出して使います。

●プログラムのしくみ

```
プログラムの実行
 ＊main関数から実行する
int main() {
    printf("Woo!");
    my_func1();
}                               main 関数

void my_func1(void) {
    :
    my_func2(100):
}                               自作関数 my_func1

int my_func2(int d) {
    :
}                               自作関数 my_func2

printf() {
    printf 関数の定義
}                               printf 関数

それぞれの関数を呼ぶ
```

つまり、プログラムは「関数のあつまり」であることを、おぼえておいてください。

5 プログラムファイルはどうやって保存する？

C言語で書いたプログラムは、「C言語のファイル」として保存しておきます。プログラムは基本的に人間が手でポチポチと書くものですから、普通にテキストエディタで書いて保存します。あとから中身を変更したい場合も、テキストエディタで編集します。

オリエンテーションで説明したように、C言語のプログラムファイルの拡張子は「.c」です。ファイル名は半角英数文字を使って自分で自由につけることができますが、全角文字や半角カナ文字は使わないようにしましょう。

プログラムファイル名は、テストをするための小さいプログラムであれば、「test.c」や「sample.c」など、適当な名前をつけてかまいません。しかし、ずっと保存しておきたいプログラムファイルは、そのプログラムでどんな処理を行っているかがわかる名前をつけておきましょう[*9]。

ヒント

*9：本書では、最終的なプログラムのみ、「janken.c」などと処理内容でファイル名をつけています。それ以外では、第3日1時限目のプログラムだったら、「3-1.c」と番号でファイル名をつけています。

例えば、文字列「Hello」だけを出力するプログラムでは、

```
hello.c
```

や、

```
printHello.c
```

とつけておくと、あとでファイル名を見ただけで、何のプログラムなのかがわかります。

C言語のプログラムファイルはどこに保存してもかまいません。本書では「C:¥source」ディレクトリにまとめて保存します。「ディレクトリ」とはファイルをまとめて保存しておく入れ物で、「フォルダ」と同じものと思ってください。以降、本書では「フォルダ」ではなく「ディレクトリ」を使います。

6 コンパイル

プログラムソースファイルを作成したら、それから実行形式のファイル（拡張子「.exe」）を作成します。これをコンパイルといいますが、この作業はCコンパイラで行います。

コンパイルを実行するのに、コマンドプロンプトから次のような命令を書きました。

```
C:¥source>gcc -o sample sample.c
```


gcc[*10]はCコンパイラプログラムです。

コマンドプロンプトでは、コマンド（命令）を入力して［Enter］キーを押せば、そのコマンドが実行されます。上記の場合は、Cコンパイラである「gcc」がコマンドです。

gccのあとにそれぞれひとつ以上の半角スペースを空けて、「-o」というオプション、実行形式ファイル名、コンパイルするファイル名を書きます。

このように、コマンドのあとにひとつ以上の半角スペースを空けて続けて何かを書いた部分は、そのコマンドに渡されます。これを引数（ひきすう）と呼びます。

gccコマンドでは、次のように-oオプションをつけずに、Cのソースファイル名だけを引数に指定にすることができます。

```
C:¥source>gcc sample.c
```

すると、きまってa.exeという名前の実行ファイルが作成されます。

-oオプションに続けて実行形式のファイル名を指定すると、その名前に拡張子「.exe」をつけたものが、実行ファイルとして作成されます[*11]。a.exeもsample.exeも、中身は同じものです。

本書では、作成する実行ファイルを、C言語のプログラムソースファイルと拡張子だけが異なるファイル名になるように指定して、コンパイルを行います[*12]。

ヒント
*10：gccはgcc.exeと入力しても同じです。コマンドプロンプトでは実行ファイルを実行するときに、拡張子「.exe」を省略することができます。

ヒント
*11：-oオプションで指定する実行ファイル名は、sample.exeと拡張子まで指定しても結果は同じです。-oオプションに拡張子がついていない場合のみ、拡張子「.exe」がつきます。

ヒント
*12：コンパイラの種類によっては、自動的にプログラムファイル名の拡張子を.exeにしたものが、実行ファイル名になるものがあります。

7 コンパイルエラーの場合

コンパイルとは、ソースコードを機械語に変換した実行ファイルを作り出す作業です。

機械語に変換するには、正確なソースコードを書く必要があります。正確なソースコードを書かなかったり、引数のファイル名の指定を間違ったりすると、「コンパイルエラー」となり、実行ファイルは作成されません。ここでは、エラー発生のよくある原因と、その対処方法を紹介します。

(1) ソースコードミスによるエラー

正確なソースコードを書いていないと、コンパイルエラーになります。例えば、sample.cの4行目の最後にセミコロンを忘れた場合、コンパイルエラーになります。

```
C:¥source>gcc -o sample sample.c
sample.c: In function `main':
sample.c:5: error: syntax error before "return"
```

「error」と表示されたのでコンパイルは正常に実行されず、実行ファイルも作成されません。コンパイルエラーのときは、たいていはエラーの原因を表示してくれるので、それをもとにプログラムを修正しましょう。

```
sample.c:5: error: syntax error before "return"
```


この例では、sample.cの5行目に対して、エラーのコメントが表示されています。

エラーの原因は、指摘された行ではなく、その直前やもっと前に原因となる箇所が存在する場合があります。指摘されたエラー行から順にさかのぼって、チェックしましょう。

もうひとつ、別のエラーを作ってみます。sample.cの4行目のprintf関数を、「print」と書いてみます。

```
print("Hello!");
```

コンパイル結果は次のとおりです。

```
C:¥source>gcc -o sample sample.c
C:¥DOCUME~1¥(略):sample.c: undefined reference to `print'
```

これは、「'print'関数なんて知らないよ」といわれています。

コンピュータプログラムは、関数名をひとつ間違えただけでもコンパイルエラーとなります。たいていはタイプミスによるエラーなので、落ち着いてプログラムを見直して修正しましょう。

(2) gccへのファイル名指定ミス

Ｃ言語のプログラムファイル名の指定ミスもあります。

```
C:¥source>gcc -o sample smple.c
gcc: smple1.c: No such file or directory
gcc: no input files
```

この例では、本来「sample.c」と指定するところが「smple.c」となっているため、エラーが発生しています。正確なファイル名を指定しましょう。

(3) ファイルのインクルード忘れミス

１行目のインクルード文はprintf関数を使うためのおまじない、と説明しました。しかし、この１行目のインクルードを書き忘れても、実行ファイルは作成されてしまいます。

そこで、-Wallというオプションをつけて実行してみましょう。これは、警告をすべて表示するオプションです。次の例は、１行目にインクルード文を書かずに、sample.cをコンパイルしようとしたものです。

```
C:¥source>gcc -Wall -o sample sample.c
sample.c: In function `main':
sample.c:4: warning: implicit declaration of function
`printf'
```

警告とは、プログラムのどこかに異常があるので修正することを推奨するメッセージです。警告が出た場合もエラーと同様に考えて、速やかにプログラムの異常箇所を見つけ出して修正しましょう[*13]。

コンパイルエラーが出た場合は実行ファイルが作成されないので、それぞれ原因を突き止めて修正します。

修正したら、再度コンパイルを実行します。それでもエラーが出るようなら、繰り返し、エラー箇所を潰していきます。この作業をデバッグといいます。

ヒント

*13：本書の以降の説明では省略しますが、できたら常に-Wallオプションをつけてコンパイルしましょう。

(4) 最後の行の改行し忘れ

次の表示が出る場合があります。

```
C:¥source>gcc -o sample sample.c
sample.c:6:2: warning: no newline at end of file
```

warning（警告）なので実行ファイルは作成されていますが、気になる人は、プログラムの最後の行の「}」のあとに、改行を入れてください[*14]。

ヒント

*14：他のＣコンパイラでは、最後の行のあとに改行がなくても、warningが出ないものもあります。

8 プログラムの実行

実行ファイルの実行方法について説明します。

(1) コマンドプロンプトからの実行

コンパイルに成功すると、プログラムの実行ファイルが作成されます。実行ファイルは、

```
C:¥source>sample.exe
```
←「sample.exe」のみ入力する

で実行しましたが、

```
C:¥source>sample
```
←「sample」のみ入力する

としても同じです*15。

sample.cは、文字列「Hello!」を出力するプログラムなので、実行したコンソール上に、

```
Hello!
```

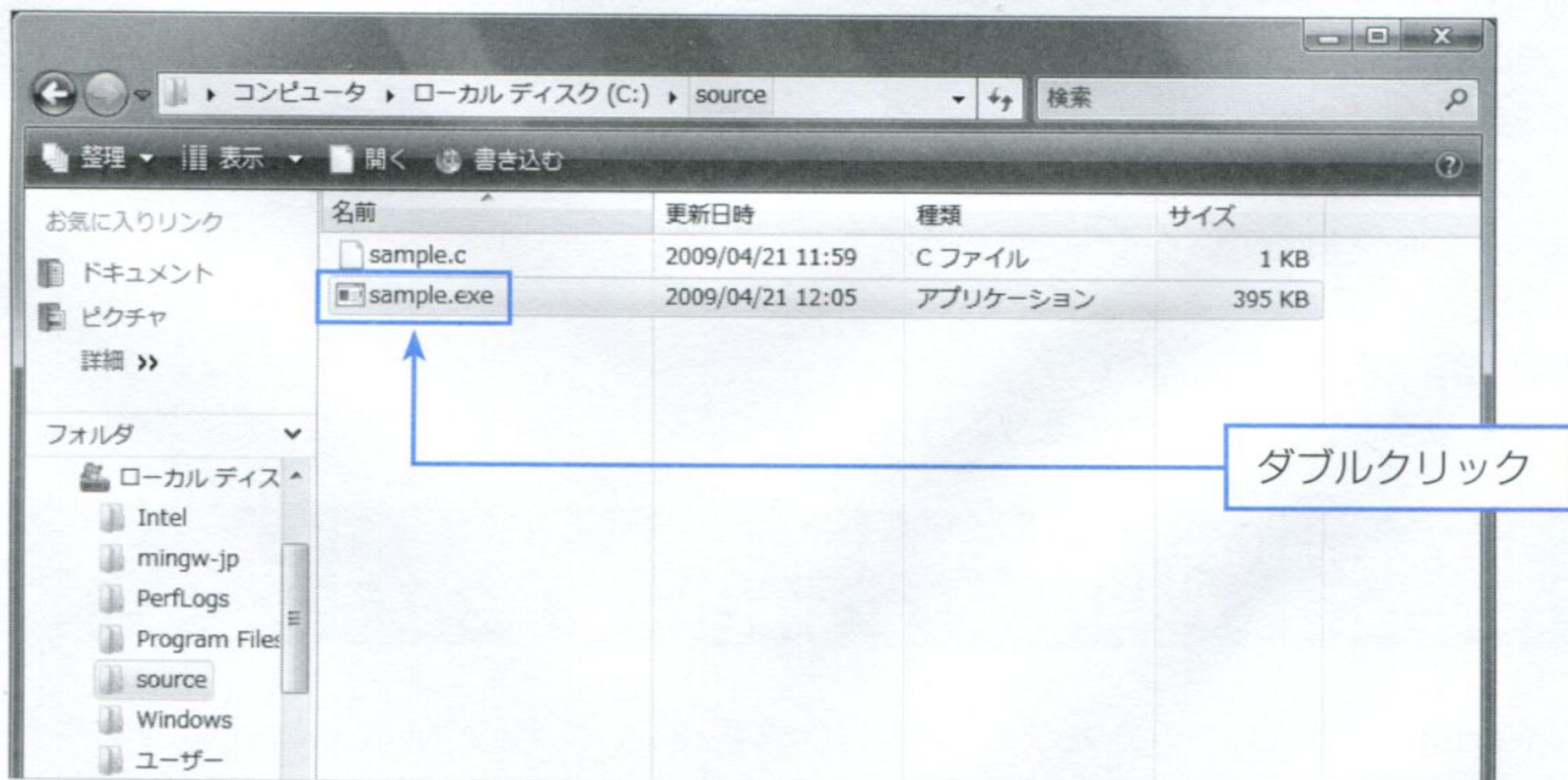
と出力されていれば成功です。

ヒント
*15：実行ファイルは拡張子が「.exe」ときまっているので、拡張子を省略しても、sample.exeが実行されます。

(2) エクスプローラからの実行

作成したのは拡張子が「.exe」の実行ファイルですから、実は、他のGUIのソフトウェアと同様にアイコンをダブルクリックして実行することもできます。

エクスプローラから「sample.exe」を実行してみてください。

●sample.exeをダブルクリックして実行

おそらく一瞬にしてコマンドプロンプト画面らしきものが立ち上がり、すぐに終了してしまったことと思います。このプログラムは、プログラムのすべての命令を処理しおわると、自動的に終了してしまうのです。したがって、コマンドプロンプトが一瞬立ち上がり、

プログラムの中身を実行して（文字列「Hello!」を出力して）終了し、すぐにコマンドプロンプト画面が終了してしまいます。

これでは納得いかない場合、プログラムを以下のようにかえてみましょう[*16]。

ヒント

*16：行番号は入力しないでください。

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello!");
5      rewind(stdin);  ←追加行：標準入力を初期化する
6      getchar();  ←追加行：1文字入力を待つ
7      return 0;
8  }
```

新しく5行目と6行目の命令文を追加することにより、エクスプローラから実行しても、何か1文字入力するまで、画面が終了することはありません。［Enter］キーを押せば画面は終了します。

しかし、本書では基本的にコマンドプロンプトからコンパイル→実行することを前提としているので、この方法は以降のプログラムには使いません。参考程度におぼえておきましょう。

まとめ

はじめてC言語プログラムを作って、動かした気分はどうですか？ エラーが出た人も無事に解決できたでしょうか？

sample.exeも立派な「ソフトウェア」です。これができたということは、コンパイラ環境も無事に整った証拠です。安心してC言語の学習を進めましょう。

練習問題

Q

sample.cを変更し、「C言語の勉強をはじめるよ！」という文字列を表示するプログラムに書き換えなさい。

解答は巻末に

第1日 2時限目

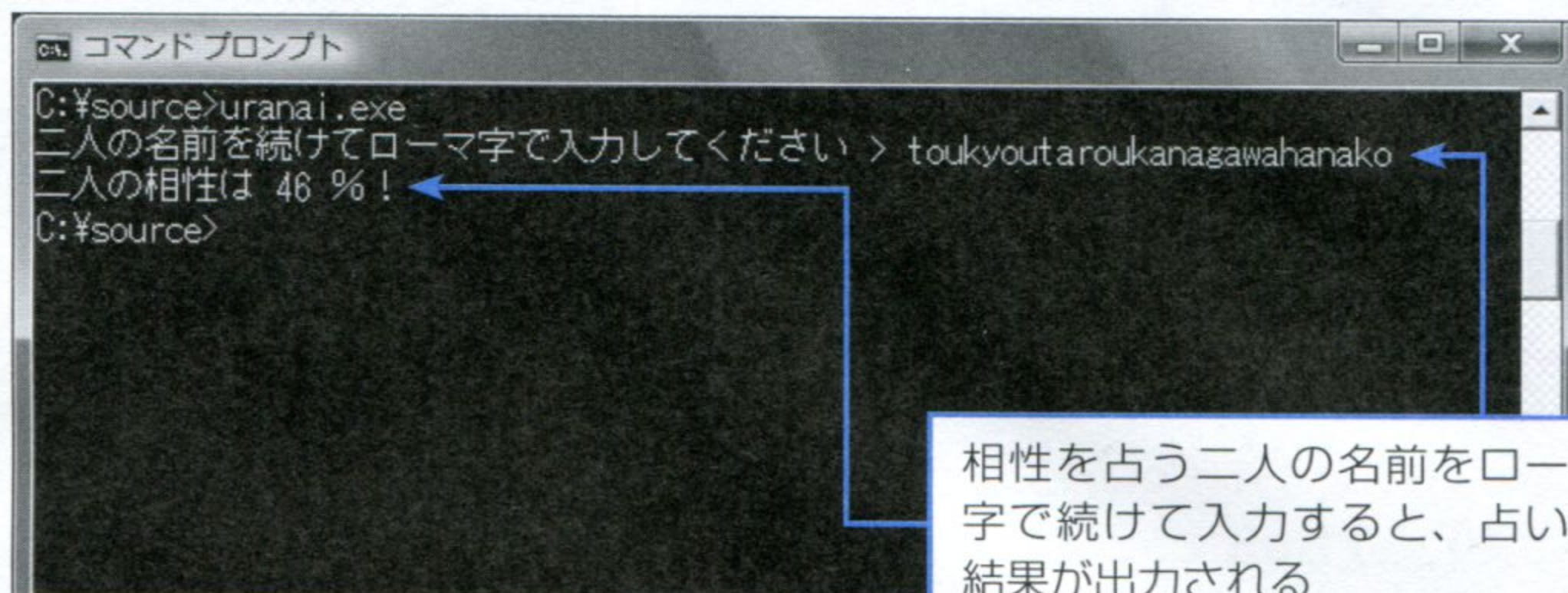
相性占いプログラムを作ろう

【はじめてのC言語プログラムを作ろう②】

やっとプログラムを書きはじめましたが、ゲームプログラムを書くには、まだ少しだけ早いかもしれません。もうちょっとプログラムを書くことに慣れましょう。
ここでは、1時限目よりも少しだけ長めのプログラムを書いてみます。

今回作成する例題

```
C:¥source>uranai.exe
二人の名前を続けてローマ字で入力してください > toukyoutaroukanagawahanako
二人の相性は 46 %!
C:¥source>
```

相性を占う二人の名前をローマ字で続けて入力すると、占いの結果が出力される

サンプルファイルはこちら 10days_c ▶ day01-02 ▶ uranai.c

●このレッスンのねらい

ここでは、まだプログラムの中身を理解する必要はありません。少し長めのプログラムを正確に書き写し、きちんと動くプログラムが作れるかどうか、挑戦してみましょう。
長めのプログラムを写していると、だんだんと書き方のコツがわかってくると思います。本レッスンの最後に、そのコツをいくつか紹介します。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する[*1]

```
#include <stdio.h>

int main() {
	char c;
	int i = 0, j;
	int d, k;
	int data[100];

	printf("二人の名前を続けてローマ字で入力してください > ");
	while ((c = getchar()) != '¥n') {
		if(c == 'a') { d = 1; }
		else if(c == 'i') { d = 2; }
		else if(c == 'u') { d = 3; }
		else if(c == 'e') { d = 4; }
		else if(c == 'o') { d = 5; }
		else { continue; }
		data[i++] = d;
	}
	if(i < 2) { printf("相性が占えません"); return 1; }

	for(j = 0; j < (i-2); j++) {
		for(k = 0; k < (i-j-1); k++) {
			data[k] = (data[k] + data[k+1]) % 10;
		}
	}

	printf("二人の相性は %d%d %！", data[0], data[1]);
	return 0;
}
```

ヒント

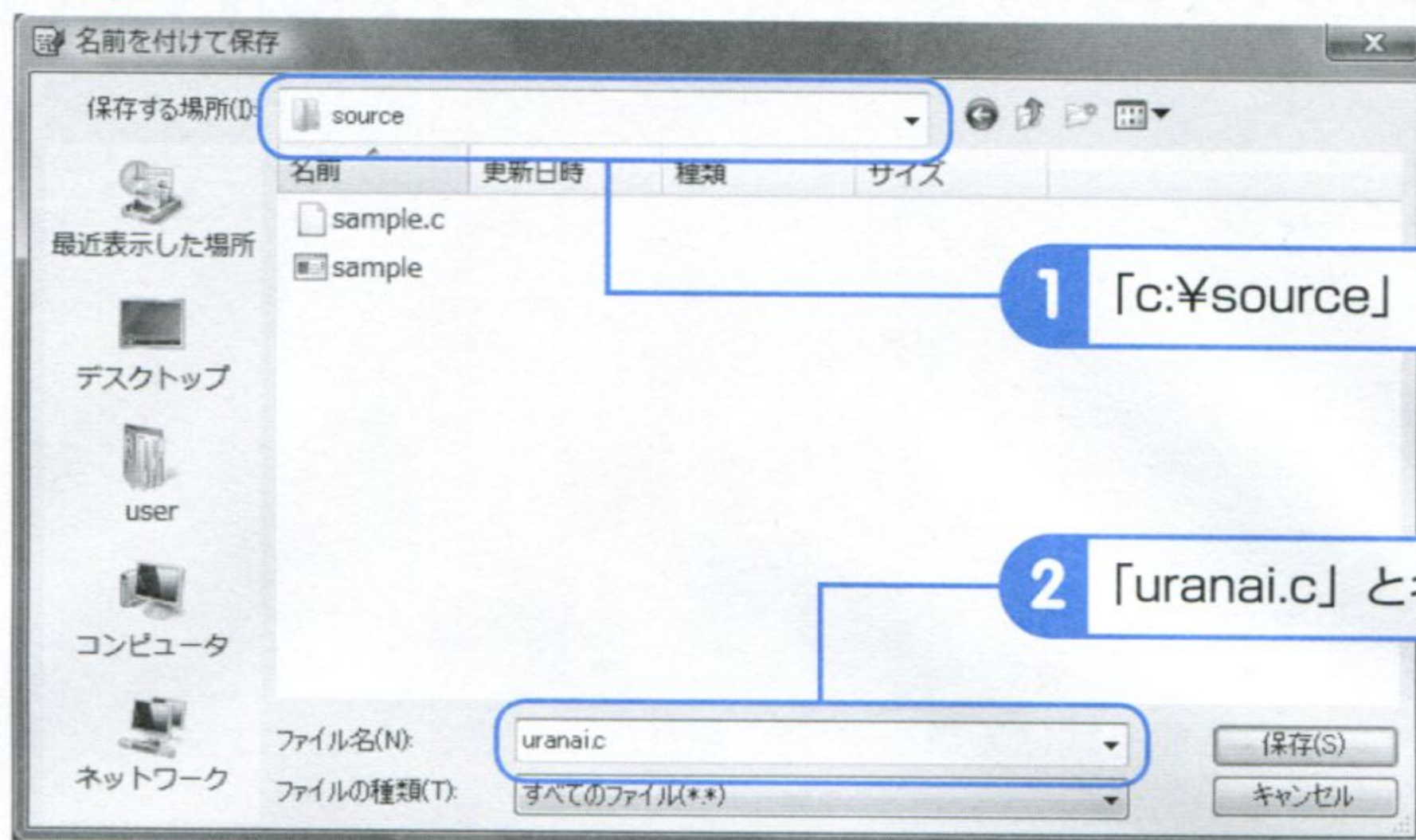
*1：インデントにはタブ（[Tab] キー）を使ってください。

ヒント
*2：拡張子に注意して保存しましょう。

2 入力できたら、「uranai.c」という名前*2で、「C:¥source」ディレクトリ下に保存する

名前を付けて保存
保存する場所(I): source
名前 更新日時 種類 サイズ
sample.c
sample
最近表示した場所
デスクトップ
user
コンピュータ
ネットワーク
ファイル名(N): uranai.c
ファイルの種類(T): すべてのファイル(*.*)
保存(S)
キャンセル

1 「c:¥source」に……

2 「uranai.c」と名前をつけて保存

ヒント
*3：プログラムが少々長いので、写し間違えてコンパイルエラーが出るかもしれません。エラーが出たら、コードをよく見直しながら修正して、再びコンパイルしましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、uranai.cをコンパイルする*3

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o uranai uranai.c
```

4 プログラムを実行する

```
C:¥source>uranai.exe
```

▼

```
二人の名前を続けてローマ字で入力してください >
```

5 相性を占う二人の名前をローマ字で続けて入力する。[Enter] キーを押すと、二人の相性結果が表示される

例）東京 太郎　　神奈川 花子
toukyou tarou　kanagawa hanako
↓
toukyoutaroukanagawahanako ← 入力

```
二人の名前を続けてローマ字で入力してください > toukyoutaroukanagawahanako
```

▼

```
二人の相性は 46 %!
```

解説

1 相性占いプログラムの内容

相性を占う二人の名前を続けてローマ字で入力して、相性を占うプログラムを作ります。今の段階ではプログラムの中身を理解する必要はありません。

簡単に内容を説明すると、まず、占う二人の名前は続けて1回で入力します。

占い方法は、名前にある母音のみを「a→1」「i→2」「u→3」「e→4」「o→5」の数値に置き換え、となりあう数値を足していき、10以上になった場合は1桁目の数値を使うことにします。足しおわったら、また同じようにとなりあう数値を足していき、最後に2つの数値になったら終了で、それが占いの結果になる、というものです。

```
t o u k y o u t a r o u k a n a g a w a h a n a k o
 5 3     5 3   1   5 3   1   1   1   1   1   1   5   ← 母音を数値に置き換える
  8   8   8 4   6 8 4   2   2   2   2   2   6        ← となりあう数値を足す
    6   6   2   0   4   2   6   4   4   4   4   8    ← となりあう数値を足す
                  (略)
                6   0   8   0                              :
                  6   8   8
                    4   6                            ← 2つの数値になったら終了
```

占い結果はまったく根拠のないものですが、子供のときにやったことのある人もいるでしょう。今回はこのプログラムを作ります[*4]。

ヒント
*4：占う名前の母音の合計が2文字よりも少ない場合は、占いを行わない処理も入れています。

2 長いプログラムを書くコツ

プログラムのmain()部分は、行頭をインデントせずに、次のように書いても結果は同じです。

【uranai.cのmain部分の抜粋】

```
int main() {
char c;
for(j = 0; j < (i-2); j++) {
for(k = 0; k < (i-j-1); k++) {
data[k] = (data[k] + data[k+1]) % 10;
}
}
return 0;
}
```

このように、インデントを使わずにプログラムを書いても実行結果は同じですが、この書き方だと非常に見にくいので、プログラムを書いた本人があとで見返しても、内容を理解するのが遅くなります。例えば、このプログラムでどこかの右括弧「}」が抜けているエラーが発生した場合、いったいどこが対応する左括弧なのか、すぐわかりません。デバッグ*5作業のためにも、プログラムは美しく書く必要があります。

ヒント
*5：デバッグとは、プログラムの間違いを修正する作業のことです。

短いプログラムではそれほど注意する点も多くありませんが、プログラムが長くなると、理解するのがより難しくなるものです。あとから見て理解しやすい、見やすいプログラムを書くように心がけましょう。長いプログラムを書くためには、プログラムが見やすくなるように、適度な改行やスペース、タブを利用します。そのためのコツを紹介します。

(1) ひとつの命令は1行で

プログラムはひとつの命令を1行で書くのが適当です。例えば、

```
命令1;命令2;命令3;
```

と書くと、すべてひとつの命令に見えてしまい、あとでプログラムを見た人が命令2と命令3を見落としてしまう可能性が高くなります。この場合は、

```
命令1;
命令2;
命令3;
```

と書くとよいでしょう。ただし、プログラム全体のバランスを考えて、1行にまとめてしまってもよい場合があります。

```
if(i < 2) {
    printf("相性が占えません");
    return 1;
}
```

これは、変数iに入っている値が２より小さければ、文字列を出力して関数をおわらせる処理ですが、

```
if(i < 2) { printf("相性が占えません"); return 1; }
```

と書いても意味は同じで、それほど見にくくなっているわけではありません。このように１行で書いた方が、すっきりして見えるかもしれません。

プログラムの書き方は人それぞれなので、文法上の間違いさえなければ「絶対」はありません。自分でプログラムを書くうちに、「自分はこういうときはこう書く！」というスタイルが徐々にできあがってくると思うので、それまではいろいろな人のプログラムを見て、参考にするのもよいでしょう。

(2) 空行を利用する

見やすいプログラムを書くには、空行（何もない行）を適当に使いましょう。コメントと同様、入れても入れなくてよいものですが、プログラムを見やすくするためには重要です。プログラムでprintf関数を使う場合、プログラムの最初にstdio.hのインクルード文を入れます。このとき、インクルード文の次の行にmain関数を書いても、文法上、問題はありません

【続けて書いた場合】

```
#include <stdio.h>
int main() {
```

しかし、インクルード文は関数を使うための準備であり、main関数はプログラムの本体です。インクルード文とmain関数は役割が違うので、この間は１行以上空けておくと、プログラムが見やすくなります。

【空行を入れた場合】

```
#include <stdio.h>

int main() {
```

main関数の中でも適度に空行を入れましょう。プログラムで行う処理はいくつかの機能に分けることができるので、各機能の区切りとして空行を利用します。

●空行を利用してプログラムを見やすくする

```
int main() {
        変数を宣言する処理;
                            ←（空行）
        入力を行う処理1;
        入力を行う処理2;
                            ←（空行）
        出力を行う処理;
        return 0;
}
```

(3) 命令文中での改行

main関数では、「main()」のあとの{から}の中に書いた部分が、関数の中身になります。はじまりの左括弧「{」の前で改行するか[*6]、あとで改行するかは人によりさまざまです。どちらでも内容に違いはありません。

ヒント

*6：左括弧の前で改行した場合、ブロックのはじまりとおわりの括弧の位置が縦の位置で揃うので、括弧の閉じ忘れの心配がありません。

【左括弧のあとで改行】

```
int main() {
}
```

【左括弧の前で改行】

```
int main()
{
}
```

今度は1行が長くなってしまう命令文を考えてみます。占い結果を表示する部分を見てみましょう。

【1行で書いた場合】

```
printf("二人の相性は %d%d %!", data[0], data[1]);
```

この1行は、次のように2行に分けて書くこともできます。

【2行で書いた場合】

```
printf("二人の相性は %d%d %!",
                              data[0], data[1]);
```

どちらも意味は同じです。1行で書くには長すぎる命令の場合、適当に改行を入れてもかまいません。ただし、変なところで改行すると、当然ながらエラーになります。

【エラーになる場合】

```
printf(" 二人の相性は
              %d%d %! ", data[0], data[1]);
```

改行していいのは、「変数名」「予約語」「演算子*7」「括弧」の前後、または引数の区切りです。改行は、スペースひとつ分と同じだと考えます。

引数とは、関数に渡す値のことです。引数が複数あるときは、カンマ「,」で区切ります。

```
printf(" 二人の相性は %d%d %! ", data[0], data[1]);
```

この場合は、最初のカンマまでが出力する文字列で、カンマで句切った残りの２つは、出力する文字列に組み込んで表示する値です。最初から順に第1引数、第2引数……と呼びます*8。

コマンドプロンプトでの引数はスペースで区切りますが、関数ではカンマで区切ります。

(4) スペースを利用する

改行と同様にスペースも適当に使いましょう。これも見やすさというよりも、プログラムを書く人の好みです。

【スペースを入れずに書く】

```
for(j=0;j<(i-2);j++){
```

【スペースを入れて書く】

```
for(j = 0; j < (i-2); j++){
```

(5) タブを利用する

「{ }」を使ったブロックでは、基本的に、その中にある命令はすべてタブを利用してインデントします。

ブロックの中にさらにまたブロックが出てきた場合は、もうひとつタブを利用してブロックを下げます。

【ブロック中のブロックはさらにインデント】

```
int main() {
    int i = 12;
    if(i > 10) {
        printf("i の値は %d", i);  ←―― ２つ目のブロックなのでさらにインデント
                :
    }
    return 0;  ←―― 元のインデントに戻った
}
```

ヒント

*7：演算子とは何らかの動作を指定する記号のことです。「=」や「+」「==」が演算子です。詳細は第３日２時限目に学習しますので、今はまだおぼえなくても大丈夫です。

ヒント

*8：コマンドプロンプトでgccコンパイラを使うときにも、ファイル名やオプションを引数にしました。コマンドや関数などに渡すものを引数と呼びます。

「if(i > 10) {」のブロックのおわりを表す「}」の次の行から、また元のインデント（タブひとつ）に戻ります。

(6) タブのかわりにスペースを４つ使う

タブは、キーボードの［Tab］キーを使って入力します。通常、タブは「スペース（空白）何個分」としてエディタごとに設定してあります。したがって、他のエディタや他の人のPCでプログラムファイルを開くと、インデントの位置がずれる場合があります。

もちろん、それぞれに「スペース何個分」ときまっているので、行ごとにインデントがずれてバラバラになるわけではありませんが、スペース４つとスペース８つでは、プログラムの見た目にかなりの差が出ます。ブロックの中にブロックを多数使用していると、一番内側のブロック部分は右に寄りすぎて、１行で表示できずに途中で折り返して表示されてしまうこともあります。

これを避けるため、タブひとつにつき、「スペース４つ」を代用する場合があります。

ただし、これは複数人で共同してプログラムを書いたりする場合に多いので、自分しか使わないプログラムは、タブによるインデントで何の問題もありません。

まとめ

少し長いプログラムでも問題なく実行することができたでしょうか。

明日からはC言語の文法を学んでいく予定です。第4日を過ぎる頃には、このレッスンで作ったプログラムを見ても、難なく理解できるだろうと思います。そのときには、このプログラムを自分なりの書き方で作り直してみるのもよいでしょう。

C O L U M N

gccの使い方

gccは各種のオプションをつけて実行します[*9]。主なオプションを紹介しておきます。

●gccの主なオプション

オプション	内容
-o	実行ファイル名を指定 。指定しないとa.exeという名前の実行ファイルが作成される 使い方）gcc -o sample sample.c
-c	コンパイルのみ行う（オブジェクトファイル[*10]を生成する） 使い方）gcc -c sample.c → sample.oを生成
-g	実行ファイルにデバッグ用の情報を埋め込む 使い方）gcc -g -o sample sample.c
-Wall	警告をすべて表示 使い方）gcc -Wall -o sample sample.c
-l	リンクするオブジェクトライブラリを指定する。lに続けてリンクするライブラリを指定する。「使い方」のmは数学関数ライブラリ（libm.a）を表す 使い方）gcc -o sample sample.c -lm
-L	コンパイラ（リンカ）がライブラリを探すディレクトリを追加
-I	コンパイラがヘッダファイルを探すディレクトリを追加

ヒント

*9：gccに--helpオプションをつけて実行すると、gccの使い方とオプションの一覧を見ることができます。

ヒント

*10：オブジェクトファイルについては第10日で学習します。

第1日 3時限目

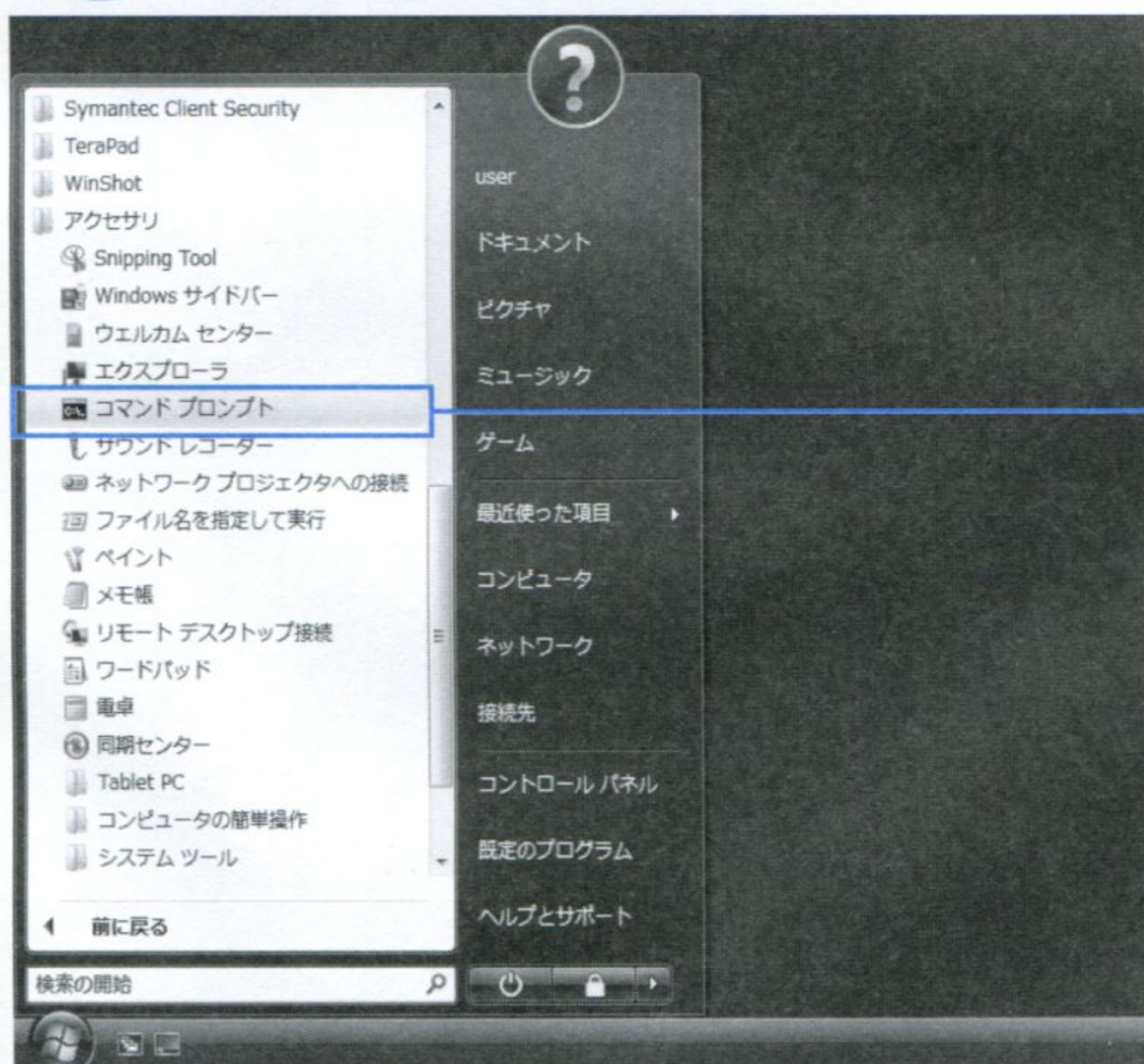
【はじめてのC言語プログラムを作ろう③】

コマンドプロンプトを使ってみよう

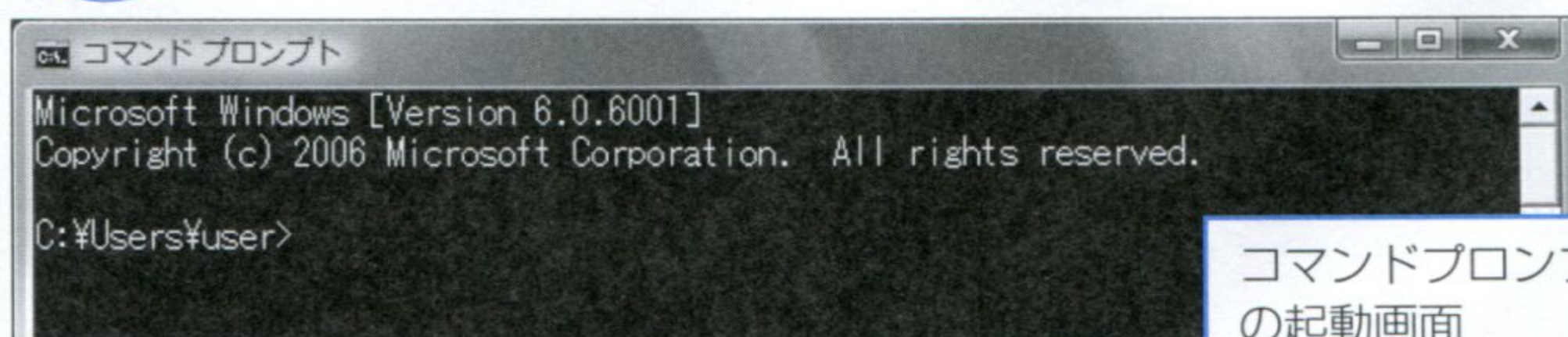
C言語で書いたプログラムは、コマンドプロンプトからコンパイルし、実行しました。
「何これ。めんどくさい……」と思った人もいるでしょう。残念なお知らせですが、本書のC言語のプログラムを実行するには、この面倒な方法を使わなければなりません。C言語のプログラムを書いたソースファイルは普通のエディタで作成しますが、そのあとのコンパイル作業とできあがった実行ファイルを動かすのは、このコマンドプロンプト画面で行います。
しかし、慣れればそれほど難しい作業ではないので、この時間にコマンドプロンプトの使い方をおぼえてしまいましょう

【注意】本レッスンのサンプルファイルはありません。

コマンドプロンプトを使う

1 「スタート」－「すべてのプログラム」－「アクセサリ」－「コマンドプロンプト」と選択し、コマンドプロンプトを起動する

1 ここをクリック

2 コマンドプロンプト上で「解説」の内容に従ってコマンドを入力し、コマンドプロンプトの操作に慣れる

```
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>
```

コマンドプロンプトの起動画面

解説

1 コマンドプロンプトとは

本書を読んでいるみなさんのほとんどは、Windowsのコンピュータを使っていると思います。

Windows上のソフトウェアの多くは、視覚的にコンピュータの操作ができるGUI（グラフィックユーザインタフェース:Graphic User Interface）です。これに対して、CUI（キャラクターユーザインタフェース：Character User Interface）とは、キーボードから文字のみで操作するソフトウェアのことをいいます。

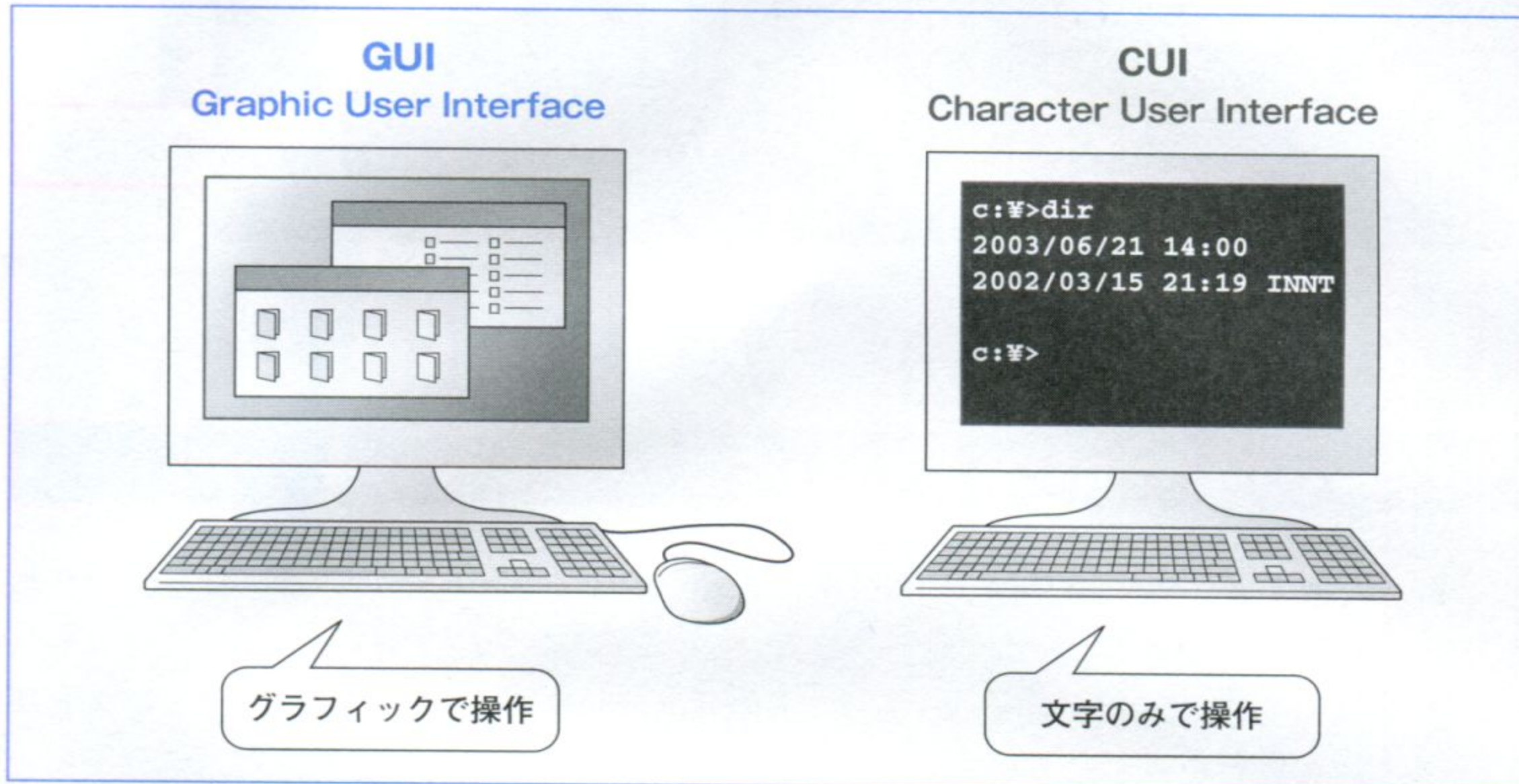
●GUIとCUI

GUI
Graphic User Interface

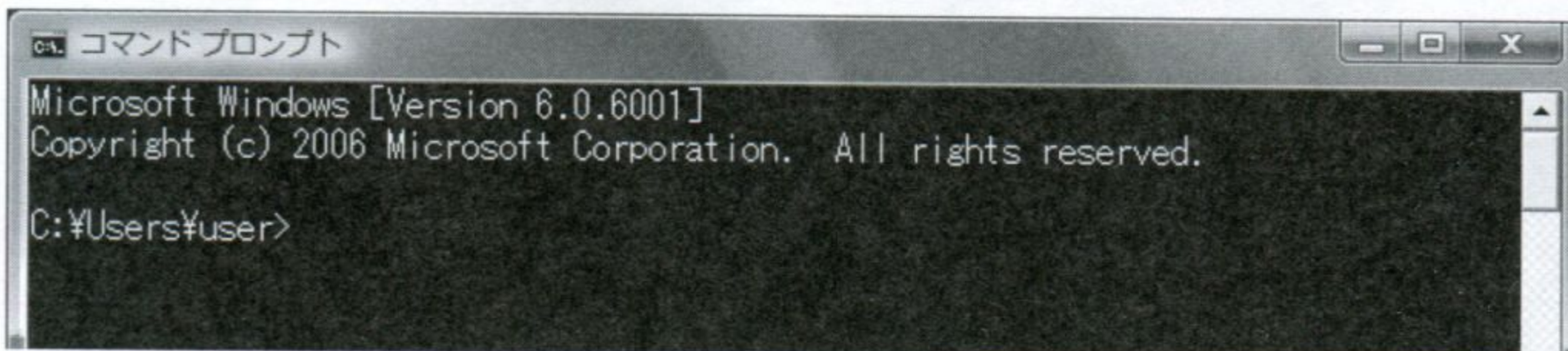
CUI
Character User Interface

```
c:¥>dir
2003/06/21 14:00
2002/03/15 21:19 INNT

c:¥>
```

グラフィックで操作

文字のみで操作

本書で作るC言語のプログラムは、すべてCUIソフトウェアになります。CUIソフトウェアは、コマンドプロンプトから実行します。「スタート」－「すべてのプログラム」－「アクセサリ」－「コマンドプロンプト」で立ち上がる画面が、コマンドプロンプト画面です。これをコンソール画面ともいいます。

●コマンドプロンプトの起動画面

```
コマンド プロンプト
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>
```

このマシンのOSはWindows Vistaで、Cドライブにインストールされています。もしもDドライブにOSがインストールされている場合は、

```
C:¥Users¥user>
```

の部分が、

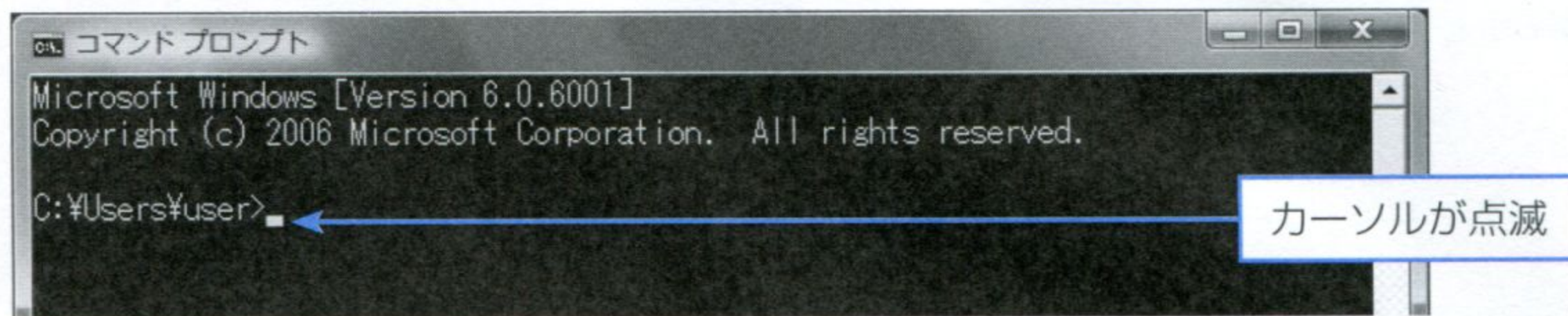
```
D:¥Users¥user>
```

となっています。

また、Windows XPでは、この部分は「C:¥Documents and Settings¥user>」となっています。Windows Vista、Windows XPのどちらも、最後の「user」部分は、ログインしているアカウントの名前です。「C:¥Windows>」と表示されるOSもあります。

コマンドプロンプト画面がフォーカスされていると、「C:¥Users¥user>」のすぐあとの部分で、カーソルが点滅しています。

●入力行でカーソルが点滅

```
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>_
```


キーボードから何か入力すると、ここに入力文字が反映されます。例えば、

```
C:¥Users¥user>dir
```

と「dir」だけ入力して[*1]、[Enter] キーを押します。すると、何やらいろいろと表示されたと思います。

これは、Cドライブの「Users」のさらに下の「user」にある、ファイルやディレクトリ（フォルダ）の一覧を表示しています。

ヒント
*1：入力するのは「dir」のみです。

●dirの実行結果（例）

```
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>dir
 ドライブ C のボリューム ラベルがありません。
 ボリューム シリアル番号は B4F1-8B7D です

 C:¥Users¥user のディレクトリ

2009/04/19  21:01    <DIR>          .
2009/04/19  21:01    <DIR>          ..
2009/04/19  21:01    <DIR>          Contacts
2009/04/21  10:37    <DIR>          Desktop
2009/04/19  21:01    <DIR>          Documents
2009/04/19  21:01    <DIR>          Downloads
2009/04/20  10:23    <DIR>          Favorites
2009/04/19  21:01    <DIR>          Links
2009/04/19  21:01    <DIR>          Music
2009/04/19  21:01    <DIR>          Pictures
2009/04/19  21:01    <DIR>          Saved Games
2009/04/19  21:01    <DIR>          Searches
2009/04/19  21:01    <DIR>          Videos
               0 個のファイル                   0 バイト
              13 個のディレクトリ  83,549,212,672 バイトの空き領域

C:¥Users¥user>
```

つまり「dir」は、「ファイルやディレクトリの一覧を表示するコマンド（命令）」です。
一覧表示の最後では、再び、

```
C:¥Users¥user>
```

と表示され、カーソルが点滅して次のコマンドを待っています。
コマンドプロンプト画面では、このようにキーボードからのコマンド入力を受けつけて実行します。コマンドの入力を待っている、

```
C:¥Users¥user>
```

の部分を、プロンプトと呼びます。先頭から「>」までがプロンプトです。
「dir」のように、コマンドプロンプト画面から特定の処理を実行させる命令文をコマンド、またはDOSコマンドといいます。

```
C:¥Users¥user> dir
      ↑          ↑
  プロンプト   コマンド
```

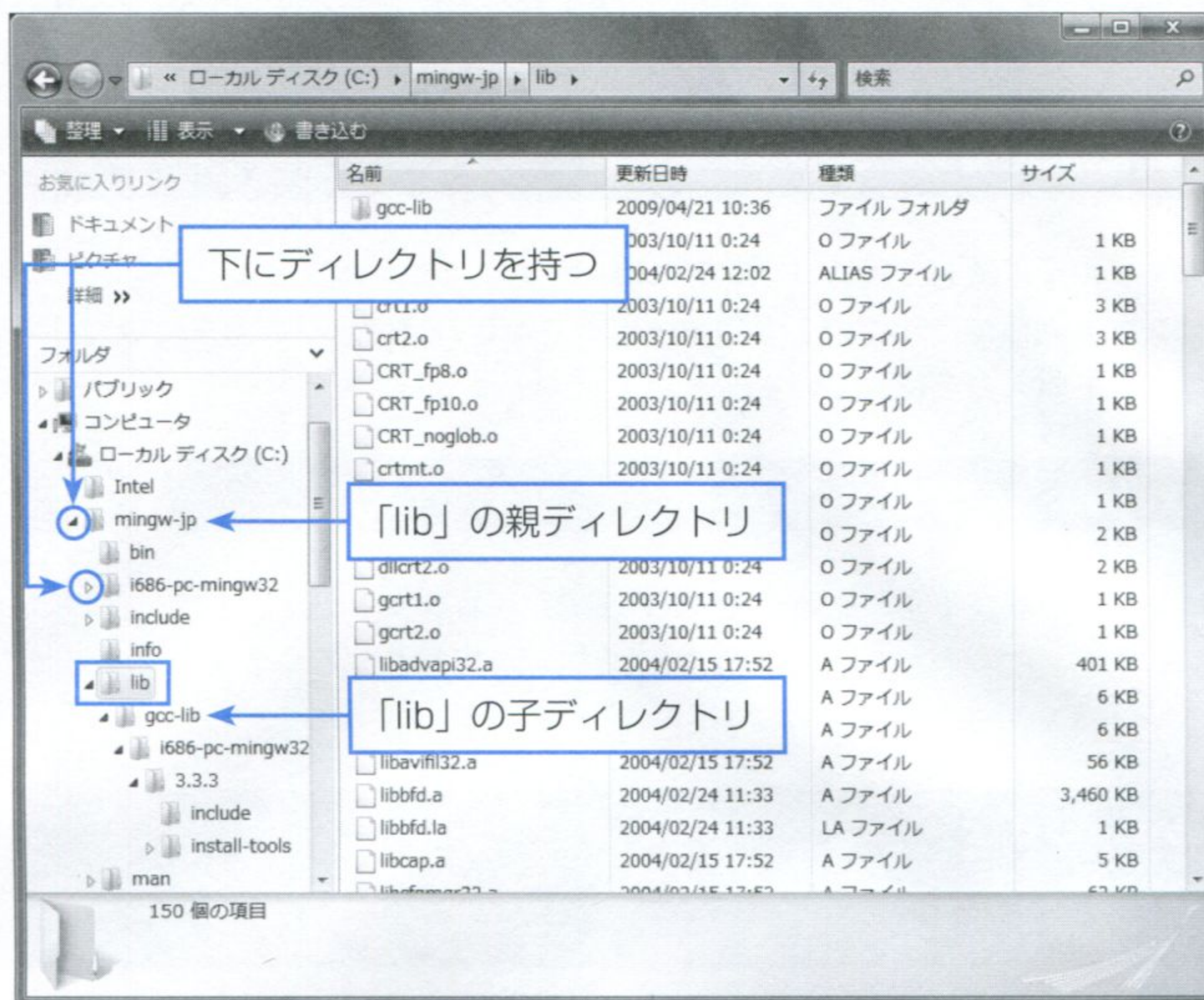
DOS*2とはDisk Operating Systemの略称でOSの一種です。Windowsが流行る前に使われていたOSで、CUIでファイルの管理などを行います。DOSを操作するために使われていたのが、dirなどのコマンドです。Windowsになってからは、Windows上からDOSコマンドが使用できます。
実は、DOSコマンドを使用するDOSプロンプトがあるのはWindows 9x系までで、Windows 2000以降のOSでは、その拡張版であるコマンドプロンプトが用意されています。本書の動作環境を考え、以降は命令文をコマンド、それを実行する画面をコマンドプロンプトで統一します。

ヒント
*2：一般的に、DOSといえばMS-DOS（Microsoft Disk Operating Systemの略）を指します。

2 ディレクトリとフォルダ

dirコマンドでディレクトリとファイルの一覧が表示されました。すでに説明したように、「ディレクトリ」とは「フォルダ」のことです*3。
スタートボタン（またはスタートメニュー）を右クリックして、エクスプローラを起動してください。エクスプローラでは、自分が今使っているPC上の、すべてのデータが表示されます。コンピュータ上のすべてのデータは、ファイルとしてハードディスクに保存されています。無数のファイルが並んでいると管理が大変なので、ディレクトリを使って階層的に管理、保存することができます。
ひとつのディレクトリの中には、0個以上のファイルやディレクトリを持つことができます。つまり、ディレクトリとは、ファイルをまとめる入れ物です。ファイルだけではなくディレクトリも持つことができ、またその下にディレクトリを……と階層構造になっています。

ヒント
*3：付属CD-ROMの構成についてのみ「フォルダ」を使用します。

●ディレクトリの階層構造

エクスプローラの左画面を見ると、ディレクトリの階層構造がわかります。エクスプローラの左画面にはディレクトリ名しか表示されません。右画面には現在選択しているディレクトリの中にある、ファイルとディレクトリが表示されます。

エクスプローラの左画面で、ディレクトリ名の左に三角形のマーク（または［+］や［−］のマーク）があるものは、そのディレクトリの下に、さらにディレクトリを含んでいることを示します。マークのないもの（上図のbinディレクトリ）の中には、ディレクトリは存在しません。

libディレクトリから見て、mingw-jpはひとつ上のディレクトリ、gcc-libはひとつ下のディレクトリです。ひとつ上のディレクトリを「親ディレクトリ」、ひとつ下のディレクトリを「子ディレクトリ」と呼びます。

コンピュータ上のデータは、すべて上下関係のディレクトリと、その中にそれぞれ含まれるファイルで構成されていることを、おぼえておいてください。

3 コマンドプロンプトからプログラムを実行

ディレクトリの概念がわかったところで、再びコマンドプロンプトの説明に戻ります。

その前に、Windowsのエクスプローラからメモ帳を起動してみましょう。メモ帳は、Cドライブ下のWindowsディレクトリ[*4]の中にnotepad.exeという名前で存在するはずです。

拡張子.exeのファイルは、実行ファイルを意味します。つまり、そのファイル単体でプログラムが実行されるファイルです。

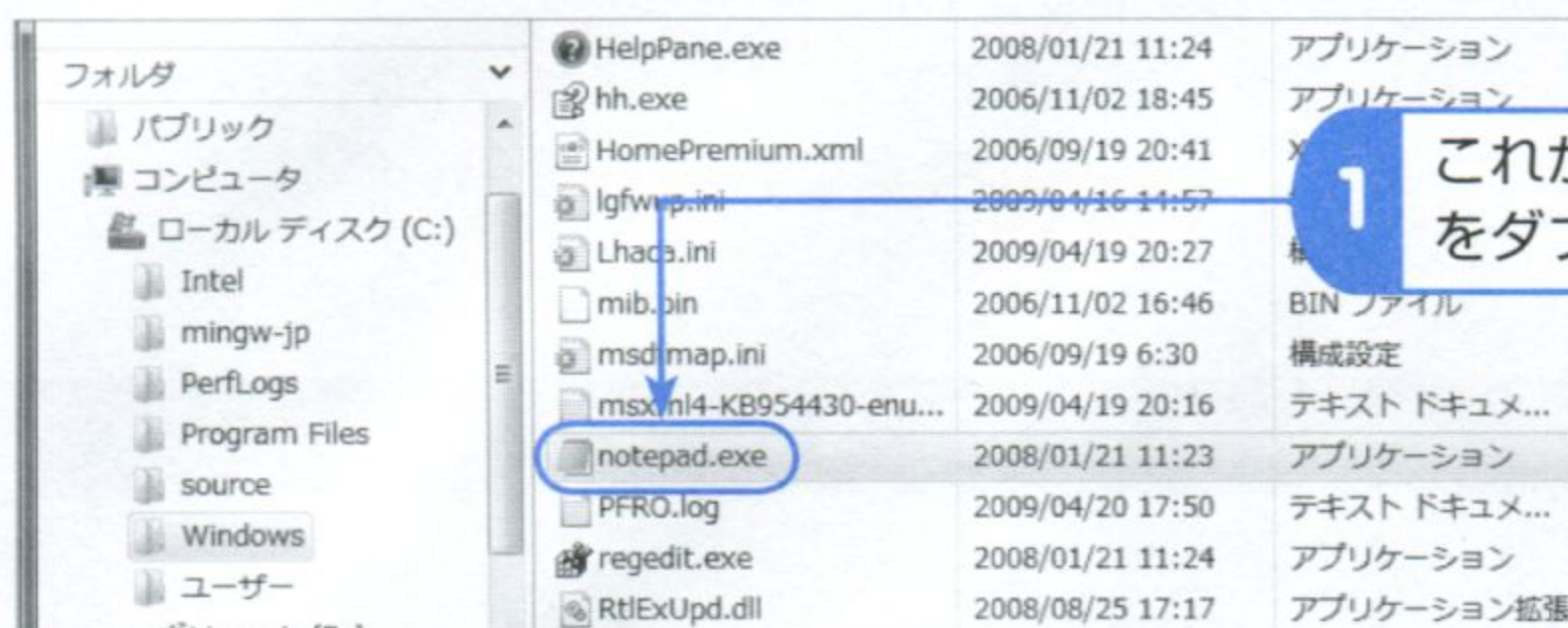
エクスプローラからnotepad.exe[*5]をダブルクリックすると、メモ帳が開きます。

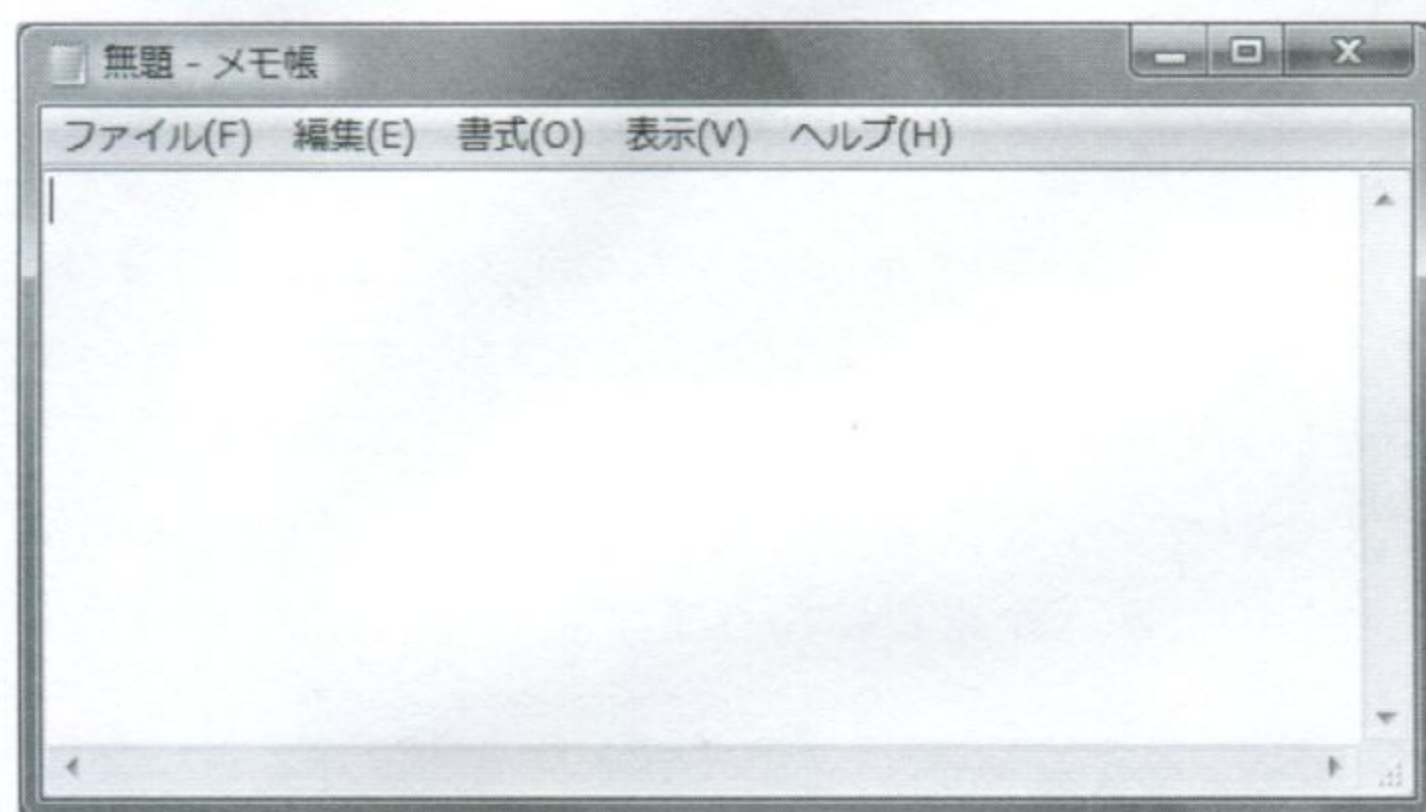
ヒント

*4：Windows環境では、基本的にファイルやディレクトリの大文字小文字を区別しないので、「WINDOWS」ディレクトリも「Windows」ディレクトリも同じです。

ヒント

*5：NOTEPAD.EXEもnotepad.exeも同じファイルです。

●メモ帳を起動する

1 これがメモ帳。「notepad.exe」をダブルクリック

●メモ帳の起動画面

つまり、「メモ帳を実行する」というのは、C:¥Windows¥notepad.exe[*6] ファイルを実行している、ということです。

ではコマンドプロンプトに戻ります。コマンドプロンプトに、

```
C:¥Users¥user>C:¥Windows¥notepad.exe
```

とメモ帳のプログラムファイル名を入力して[*7]、[Enter] キーを押しましょう。

エクスプローラからダブルクリックして実行したときと同様、メモ帳が開いたはずです。

つまり、コマンドプロンプトからは、DOSコマンドだけでなく拡張子が.exeの実行ファイルも、プログラムコマンドとして実行することができるのです。

4 ディレクトリの移動①

コマンドプロンプトでディレクトリを移動してみます。

まず、コマンドプロンプト上で「cd ¥」と入力して[*8]、[Enter] キーを押してください。

```
C:¥Users¥user>cd ¥

C:¥>
```

ヒント

*6：「¥」はディレクトリの区切りを表します。

ヒント

*7：入力するのは「C:¥Windows¥notepad.exe」のみです。

ヒント

*8：「C:¥Users¥user>」の部分の最初が「C:」でない場合は、先に「c:」と入力してドライブを変更してください。

「C:¥Users¥user>」の部分が「C:¥>」にかわりました。

現在、コマンドを実行する位置は、Cドライブの直下です。Cドライブの下には複数のファイルやディレクトリがあります。では、その中のひとつであるWindowsディレクトリに移動してみましょう。

```
C:¥>cd Windows
```

と入力して、[Enter] キーを押します[*9]。「cd」はチェンジディレクトリ、つまりディレクトリを移動するコマンドです。

ヒント
*9：入力するのは「cd Windows」のみです。「cd WINDOWS」と入力しても同じです。Windows環境では、ファイルやディレクトリ名の大文字小文字を区別しません。

これでコマンドプロンプトが、

```
C:¥Windows>
```

にかわりました。コマンドを実行する位置は、C:¥Windowsに移動したことになります。今現在コマンドを実行するディレクトリを、カレントディレクトリといいます。プロンプトに表示されているディレクトリがカレントディレクトリです。

コマンドプロンプトでは、さまざまなコマンドを入力して実行しますが、そのときの位置、つまり、どこからコマンドを実行しのたか、が重要になります。

```
C:¥Users¥user>dir ←── C:¥Users¥user のディレクトリ一覧を表示する

C:¥>cd Windows ←── C:¥ の位置からすぐ下の Windows ディレクトリに移動する

C:¥Windows>dir ←── C:¥Windows のディレクトリ一覧を表示する
```

ヒント
*10：ファイルやディレクトリ一覧の数が多いと表示が流れてしまいますが、そのときはスクロールバーを使って確認します。

ここでdirコマンドを実行すると、C:¥Windows以下にあるディレクトリやファイル名一覧が表示されます[*10]。その中に、notepad.exeが存在します。ここで、

```
C:¥Windows>notepad.exe
```

と入力して [Enter] キーを押し[*11]、実行してみましょう。

ヒント
*11：入力するのは「notepad.exe」のみです。

```
C:¥>C:¥Windows¥notepad.exe
```

を実行した時と同様に、メモ帳が開いたと思います。

5 ディレクトリの移動②

ディレクトリを移動するcdコマンドは、引数に移動したいディレクトリ名を指定します。第0日でCコンパイラのインストールが終了している場合は、mingw-jpというディレクトリが、ご使用のマシンの中にあると思います。今度はそのディレクトリに移動してみましょう。

次のように入力していきます[*12]。ここでは、まず、Cドライブ直下に戻ってから移動しています。

```
C:¥Windows>cd ¥

C:¥>cd mingw-jp

C:¥mingw-jp>
```

ヒント
＊12：入力するのは「cd ¥」と「cd mingw-jp」のみです。

mingw-jpディレクトリへの移動は、次のように書いても同じです。

```
C:¥>cd .¥mingw-jp
```

今度はプロンプトが、

```
C:¥mingw-jp>
```

にかわりました。ディレクトリが移動した証拠です。

「.」は自分自身のディレクトリ、「¥」はディレクトリの区切りを表しています。「.¥mingw-jp」は、自分のディレクトリの下のmingw-jpディレクトリ、という意味です。

次のように入力して[*13]、もうひとつ下のlibディレクトリに移動してみます。

ヒント
＊13：入力するのは「cd lib」のみです。

```
C:¥mingw-jp>cd lib

C:¥mingw-jp¥lib>
```

ヒント
＊14：入力するのは「cd ..」のみです。

今度は、ひとつ上のディレクトリに戻ってみましょう。次のように入力してください[*14]。

```
C:¥mingw-jp¥lib>cd ..
```

「..」はひとつ上のディレクトリを表します。これは、次のように入力しても同じです。

```
C:¥mingw-jp¥lib>cd ..¥
```

または、「cd .¥.」でも同じです。今度はプロンプトが、

```
C:¥mingw-jp>
```

にかわりました。libディレクトリの上にある、mingw-jpディレクトリに戻ってきました。

「.」や「..」と、ディレクトリの区切りを表す「¥」とディレクトリ名を使って、移動をいろいろと試してみましょう。

ディレクトリの移動は、2階層以上にまたがって指定することもできます。

```
C:¥mingw-jp¥lib>cd ..¥..
```

とすると[*15]、ひとつ上のさらにひとつ上、つまり、2つ上のディレクトリである「C:¥」に移動します。

今度は複数の階層をさがってみましょう。次のように入力すると[*16]、「C:¥」から「C:¥mingw-jp¥lib」へ一気に移動します。

```
C:¥>cd mingw-jp¥lib
```

次に、横に移動してみましょう[*17]。

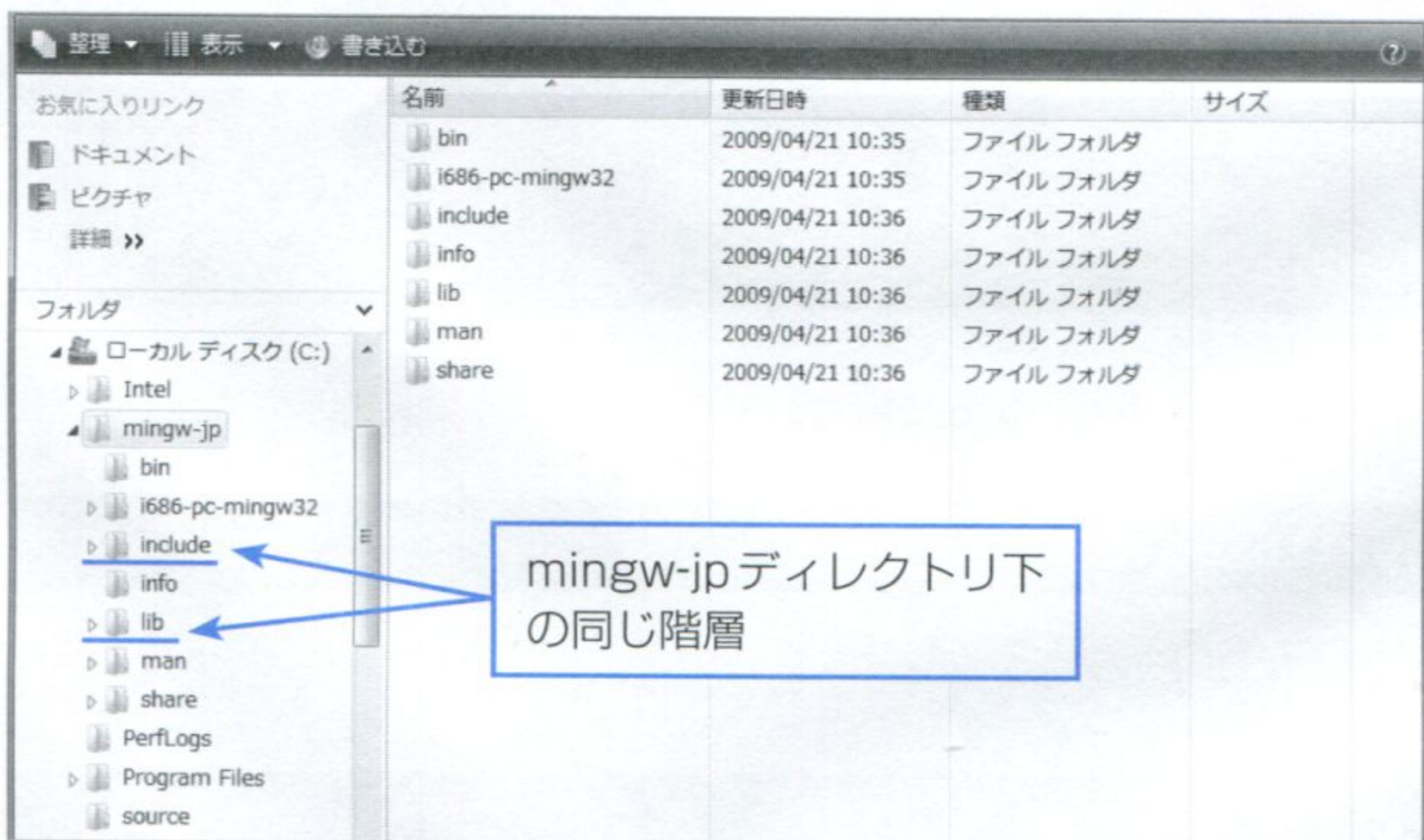
```
C:¥mingw-jp¥lib>cd ..¥include
```

この例では、「C:¥mingw-jp¥include」へ移動しました。「C:¥mingw-jp」の下の「lib」ディレクトリと「include」ディレクトリは、同じ階層にあります。

「..¥include」は、ひとつ上のディレクトリ（..¥）の下にあるincludeディレクトリを表しています。

ヒント
*15：入力するのは「cd ..¥..」のみです。

ヒント
*16：入力するのは「cd mingw-jp¥lib」のみです。

ヒント
*17：入力するのは「cd ..¥include」のみです。

●mingw-jpディレクトリ

6 相対パスと絶対パス

ファイルやディレクトリの存在する位置の表示方法を「パス」と呼びます。パスには2つの種類があります。「相対パス」と「絶対パス」です。

(1) 相対パス

cdコマンドでは、「cd」のあとに続けて、移動するディレクトリ名を引数として指定します。

今までの移動方法のほとんどは「相対パス移動」といって、カレントディレクトリから見た目的の位置を指定する方法です。

```
C:¥>cd mingw-jp
```

この場合、カレントディレクトリがC:¥のときに、C:¥から見てすぐ下にあるmingw-jpディレクトリに移動しています。

```
C:¥Windows>notepad.exe
```

とした場合は、相対パス指定でメモ帳を実行していることになります。

(2) 絶対パス

「相対パス」に対して、「¥」ではじまるパス形式を「絶対パス」といいます。

「¥」のみだと、階層構造の一番上を表しています。階層構造の一番上をルートと呼びます。つまり、絶対パスとはルートから目的の位置まで、すべて指定する形式です。

```
C:¥mingw-jp>cd ¥source
```

と指定すると[*18]、C:¥sourceディレクトリに移動します。

「絶対」指定なので、カレントディレクトリがどこであろうと、関係ありません。

Cドライブの直下、つまりルートに移動したい場合は、ディレクトリに「¥」のみを指定します。これも、この時間ですでに何度か出てきました。

```
C:¥mingw-jp>cd ¥
```

基本的に、ディレクトリの移動は同じドライブ内で行うので、ドライブ名は省略されています。明示的にドライブ名をつけて書いても同じです。

```
C:¥mingw-jp>cd C:¥source
```

上記の例では、同じドライブ（Cドライブ）内のsourceディレクトリへ移動しています。

では、次のように入力して[*19]、絶対パス指定でメモ帳を実行してみましょう。

```
C:¥>C:¥Windows¥notepad.exe
```

(3) パスの読み方

「¥」はディレクトリの区切りを表しています。パスは「¥」で区切って、前から順に読んでいきます。

```
C:¥Windows¥notepad.exe
```

ヒント

*18：これは本日の1時限目、2時限目ですでに行っています。まだカレントディレクトリの概念がなかったために、相対パスを使わずに絶対パスで移動しました。

ヒント

*19：入力するのは「C:¥Windows¥notepad.exe」のみです。

上記の場合、"「Cドライブ」直下の「Windows」下の「notepad.exe」"という意味になります。

```
..¥lib¥gcc-lib
```

これは、「C:¥mingw-jp」の下のどこかのディレクトリにいると考えて、そこから"ひとつ上のディレクトリの下の「lib」の下の「gcc-lib」ディレクトリ"を指しています。

(4) dirコマンドの引数

dirコマンドの引数にパスを指定することもできます。引数を省略するとカレントディレクトリの中身の一覧を表示しますが、引数にディレクトリを指定すると、そのディレクトリの中身を表示します。相対パス指定では、

```
C:¥mingw-jp>dir ..¥
```

これでひとつ上のC:¥のファイル一覧を表示します。

絶対パス指定では、

```
C:¥mingw-jp>dir ¥source
```

で、C:¥sourceのディレクトリの中身が表示されます。

(5) ドライブの指定

通常、PCにはドライブ[*20]が存在しています。

コマンドプロンプト上で別ドライブのディレクトリへ移動するには、

```
C:¥>cd D:¥temp
```

と指定しても、移動できません。ドライブごとにルート「¥」が存在しているので、一度ドライブを変更しなければならないのです。

ドライブを変更するには[*21]、

```
C:¥>d:

D:¥>
```

と指定します。移動するドライブ名のあとに続けて「:」を入力し、[Enter]キーを押します。あとは絶対パスでも相対パスでも、自由にこのドライブ内を移動してください。

ヒント

*20：ドライブとは、HDD、CD-ROMやDVDなど、記憶装置のことを指します。同一HDDの中でも容量を区切り、いくつかに分割して別ドライブとしてデータを管理することができます。

ヒント

*21：ドライブ名は、大文字小文字が関係ありません。移動するときは「D:」でも同じです。

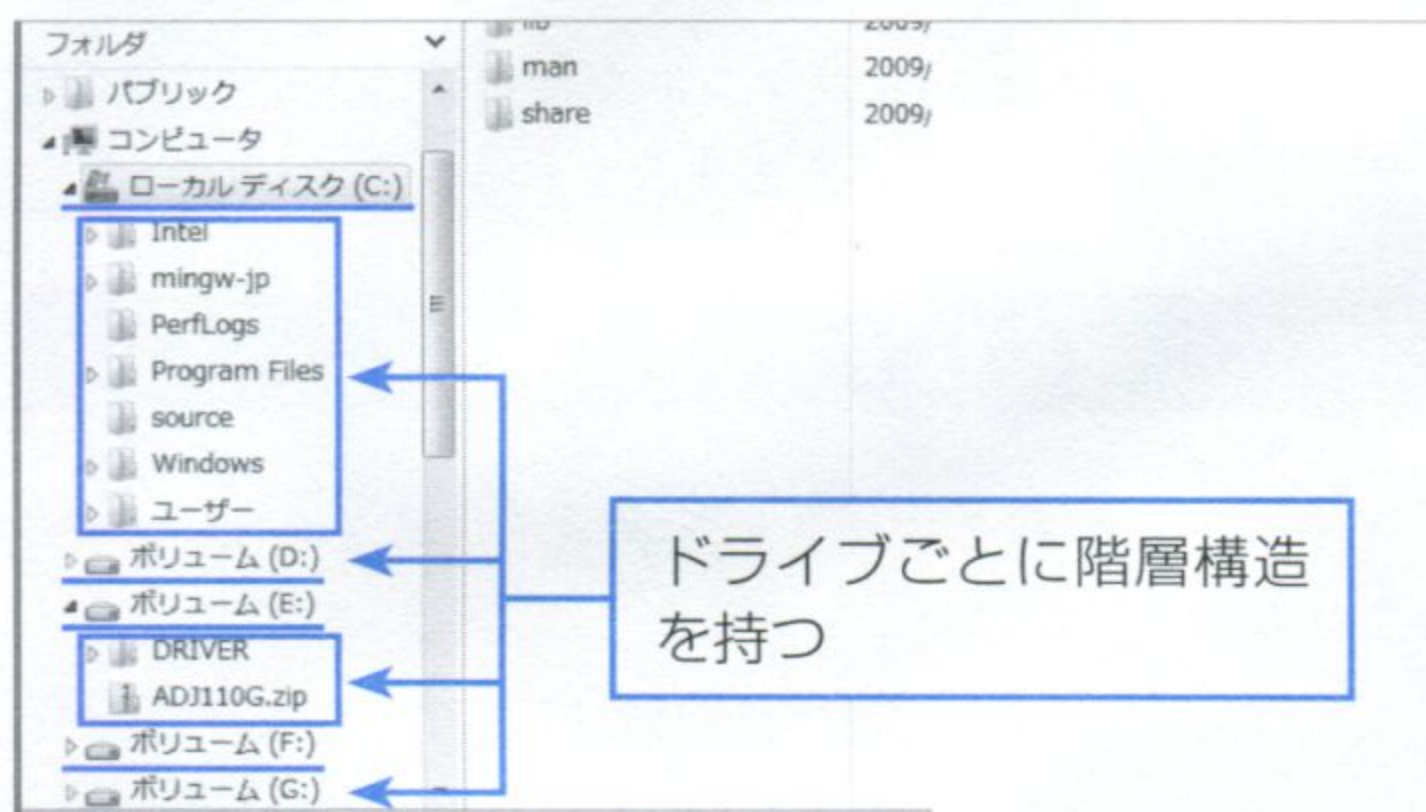
しかし、ディレクトリ情報を表示するdirコマンドの場合は、ドライブを移動する必要はありません。

```
C:¥>dir D:¥temp
```

と指定すれば表示できます。ただし、引数はドライブ名に続いて絶対パスで指定する必要があります。

コンピュータ内のデータは階層構造で管理、保存されていると説明しました。階層構造の一番上をルートと呼びますが、ひとつのドライブ内では、ルートはひとつだけです。

PC上では複数個のドライブを持つことができるので、ドライブの数だけルートが存在します。

●複数のドライブがあるPCの場合

7 環境変数について

先ほど、コマンドプロンプトからC:¥Windowsディレクトリの下にあるnotepad.exeを実行しました。C:¥Windowsディレクトリまで移動してそこからnotepad.exeを実行する方法と、他のディレクトリからC:¥Windows¥notepad.exeと絶対パスで指定して実行する方法と、両方を試したと思います。

しかし、実はnotepad.exeに関してはカレントディレクトリがどこであっても、

```
C:¥Users¥user>notepad.exe ←「notepad.exe」のみ入力
```

で実行することができます。

これは、notepad.exeの存在するディレクトリに環境変数Pathが通っているからです（以降、「Path」は「PATH」[*22]と表記します）。

環境変数PATHの値は、Cコンパイラのインストール後に設定を行いました。おぼえているでしょうか。

環境変数PATHの値は、コマンドプロンプトからも確認できます。どこのディレクトリからでもいいので、次のechoコマンドを実行してください。

ヒント

*22：PATHは「パス」と読みます。相対パスや絶対パスの「パス」と区別するためにも，（環境変数）PATHと書きます。

```
C:¥Users¥user>echo %path%  ←「echo %path%」のみ入力
C:¥Windows¥system32;C:¥Windows;C:¥Windows¥System32¥
Wbem;c:¥mingw-jp¥bin...
```

いろいろなディレクトリの絶対パスが、セミコロン「;」でつながれて表示されたと思います。この各ディレクトリの中にある実行ファイルは、カレントディレクトリがどこであっても実行できるのです。この状態を「PATHが通っている」といいます。

インストール後に行った作業は、この「PATHを通す」作業でした。Cコンパイラの実体であるgcc.exeファイルが存在するディレクトリ「C:¥mingw-jp¥bin」を、gcc.exeのPATHとして、環境変数に追加したのです。

この設定で、Cコンパイラの実体であるgcc.exeは、コマンドプロンプト画面でカレントディレクトリがどこであろうと、実行することができるようになったのです*23。

ヒント

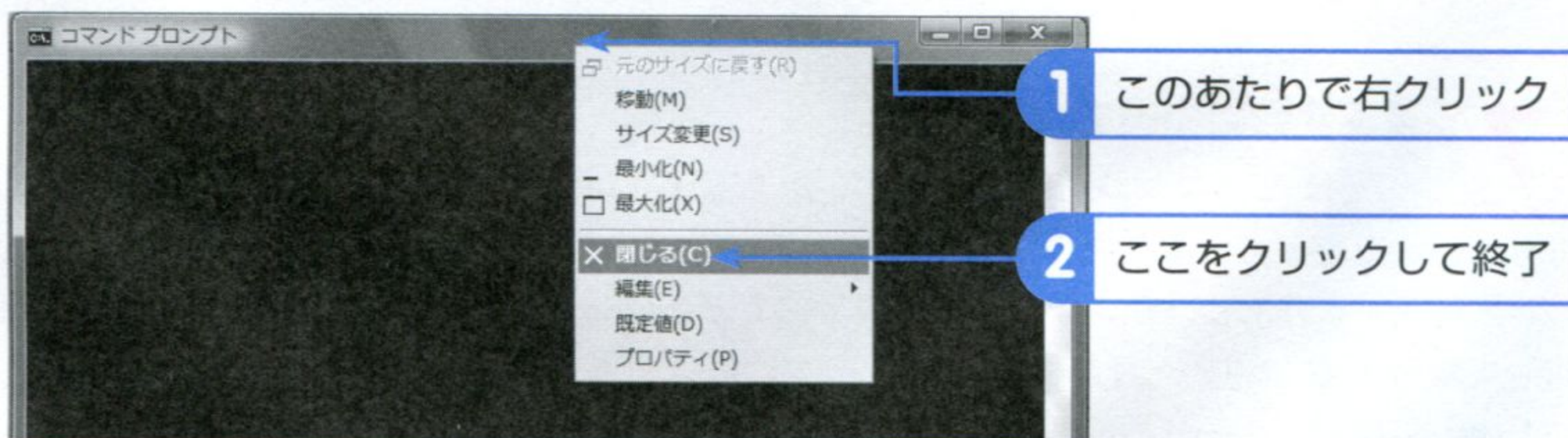
*23：第0日で作成したsample.exeやuranai.exeは、カレントディレクトリが「C:¥source」以外のときは実行できませんが、「C:¥source」を環境変数PATHに追加すれば、どこからでも実行することができます。

8 コマンドプロンプトの便利な使い方

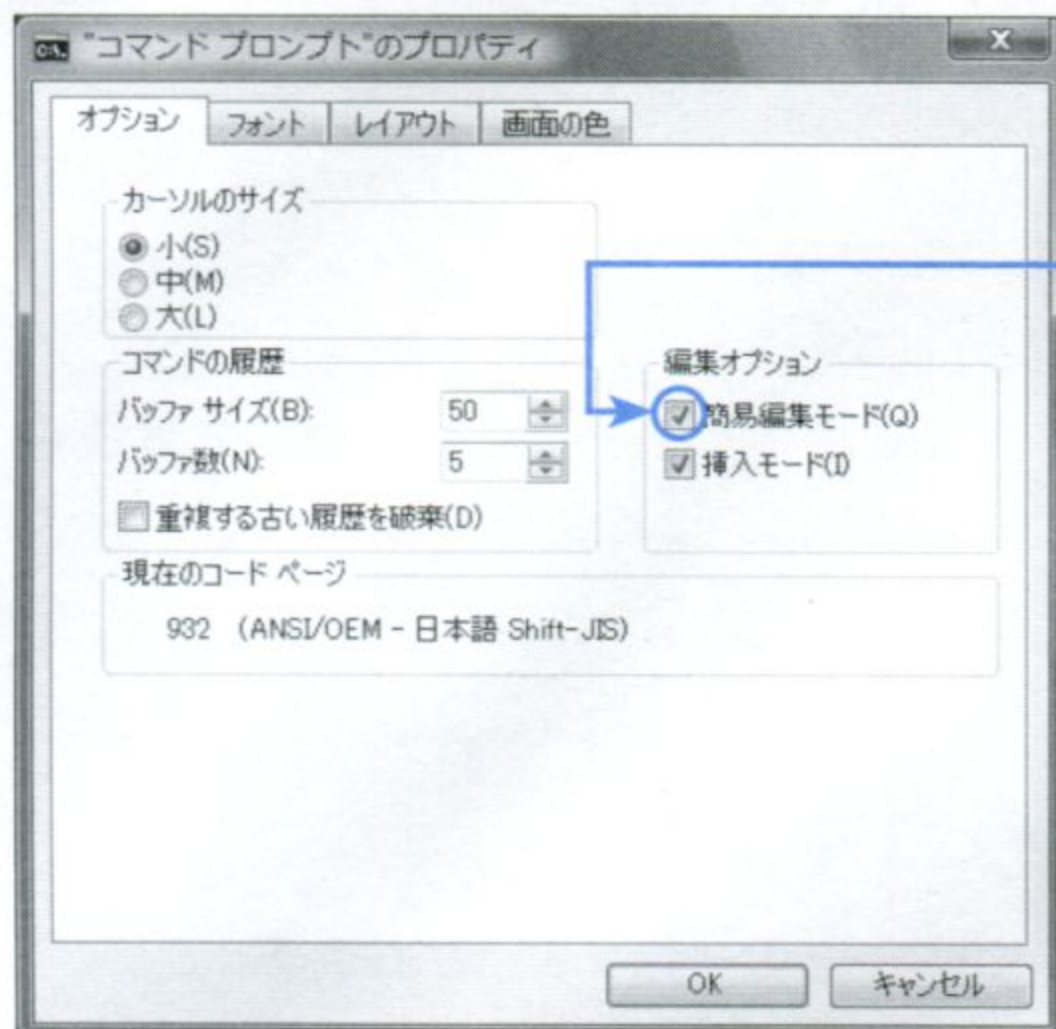
コマンドプロンプト画面を終了するには、exitコマンドを使うか、画面右上の「閉じる」ボタン（「×」ボタン）をクリックすると、終了できます。

```
C:¥Users¥user>exit
```

または、コマンドプロンプトのタイトルバーを右クリックして表示されるメニューから、「閉じる」を選択して画面を閉じることもできます。

●メニューからコマンドプロンプトを終了

コマンドプロンプトで何度も同じコマンドを打つのが面倒な場合は、コピー&ペーストを使うと便利です。タイトルバーを右クリックして表示されるメニューから「プロパティ」を選択し、プロパティ画面の「オプション」タブにある「簡易編集モード」にチェックを入れると、簡単にできるようになります。

●コマンドプロンプトのプロパティ画面

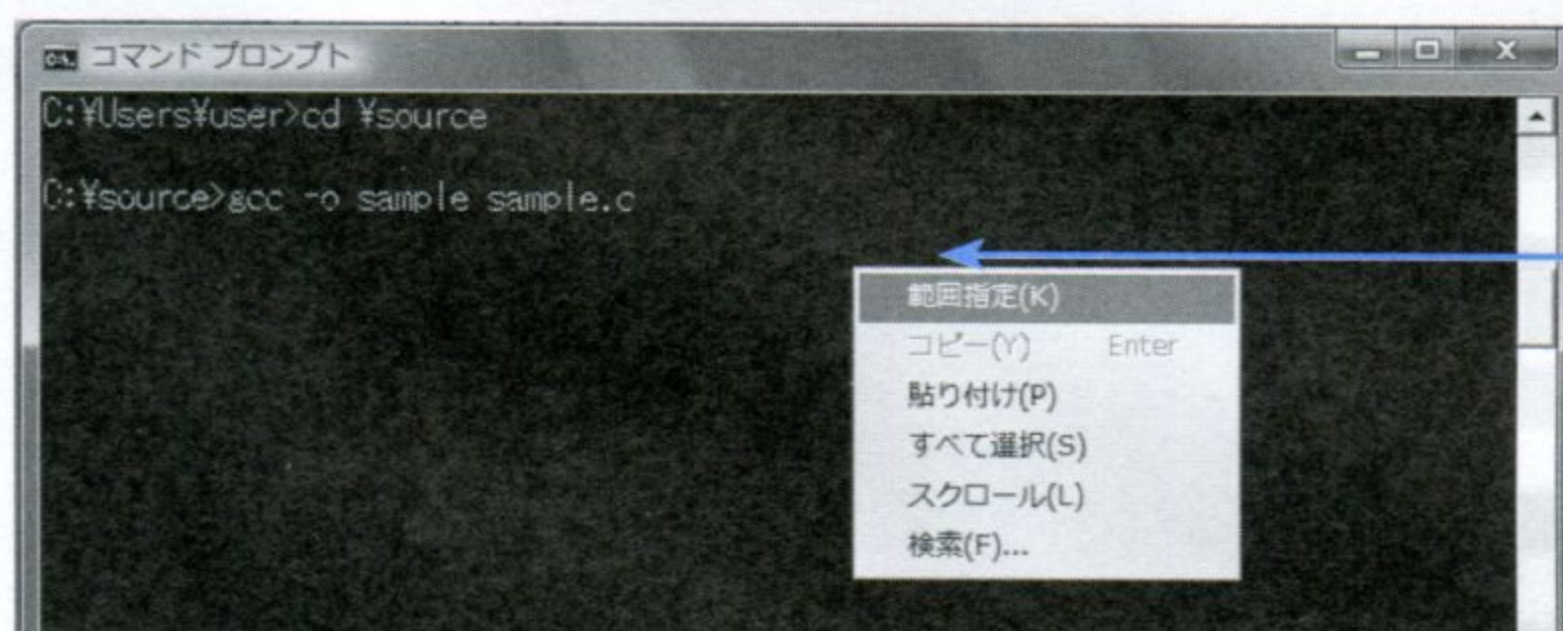
"コマンド プロンプト"のプロパティ
オプション フォント レイアウト 画面の色
カーソルのサイズ
小(S)
中(M)
大(L)
コマンドの履歴
バッファ サイズ(B): 50
バッファ数(N): 5
重複する古い履歴を破棄(D)
編集オプション
簡易編集モード(Q)
挿入モード(I)
現在のコード ページ
932 （ANSI/OEM - 日本語 Shift-JIS）
OK キャンセル

ここをチェック

対象の文字列部分をドラッグして［Enter］キーを押すと「コピー」でき、カーソルが点滅している部分でそのまま右クリックすると、「ペースト」できます。この方法はとても操作が楽になるので、簡易編集モードにチェックを入れておきましょう。

簡易編集を使わない場合は、コンソール上の適当な位置で右クリックして編集メニューを表示し、「範囲指定」を選択します。コピーしたい文字列部分をドラッグして選択し、［Enter］キーを押すと「コピー」できます。コンソール上の適当な位置で右クリックして編集メニューを表示し、「貼り付け」を選択すると、「ペースト」できます。

●コマンドプロンプトのプロパティ画面

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o sample sample.c
```

コマンド プロンプト
範囲指定(K)
コピー(Y) Enter
貼り付け(P)
すべて選択(S)
スクロール(L)
検索(F)...

右クリックして編集メニューを表示

なお、これはWindows全体で使うコピー＆ペースト機能なので、コマンドプロンプトで作業をしている際にその他のアプリケーションで何かの文字列をコピーしてしまうと、次にコマンドプロンプト上でペーストしたときには、その他のアプリケーションでコピーした文字列がペーストされてしまいます。注意しましょう。

まとめ

コマンドプロンプトについて学習しました。コマンドプロンプトの操作方法は、本書でのC言語のプログラム作成には必要な知識です。この操作に慣れておくと、この先、UNIX系のOSを使うことがあったときにも、きっと役に立つと思います。

絶対パスや相対パスの指定方法は、今後プログラムを書く上で知っておかなくてはならない重要な事柄のひとつです。さらっと流してしまった人は、よく見直してください。

COLUMN

コマンドプロンプトでよく使うコマンド

コマンドプロンプトからhelpを実行*24すると、コマンドの一覧が表示されます。ここでは、主なコマンドを紹介しておきます。

ヒント
*24：カレントディレクトリはどこでもかまいません。

```
C:¥>help
```

●主なコマンド

コマンド	使い方	意味
cd	cd ディレクトリ	ディレクトリを移動する
cls	cls	画面をクリアする
copy	copy ファイル1 ファイル2	ファイルをコピーする
del	del ファイル	ファイルを削除する
dir	dir ディレクトリ名	ディレクトリ内のファイルや子ディレクトリを表示する
echo	echo メッセージ	メッセージを表示する
exit	exit	コマンドプロンプト画面を終了する
help	help	コマンドのhelpを表示する
mkdir	mkdir ディレクトリ	ディレクトリを作成する
more	more ファイル	ファイルの中身を一画面ずつ表示する
move	move ディレクトリ1 ディレクトリ2	ディレクトリの名前を変更する
rename(ren)	rename ファイル1 ファイル2	ファイルの名前を変更する
rmdir(rd)	rmdir ディレクトリ	ディレクトリを削除する
type	type ファイル	ファイルの中身を表示する
xcopy	xcopy ファイル1 ファイル2 xcopy ディレクトリ1 ディレクトリ2	ファイルやディレクトリをコピーする

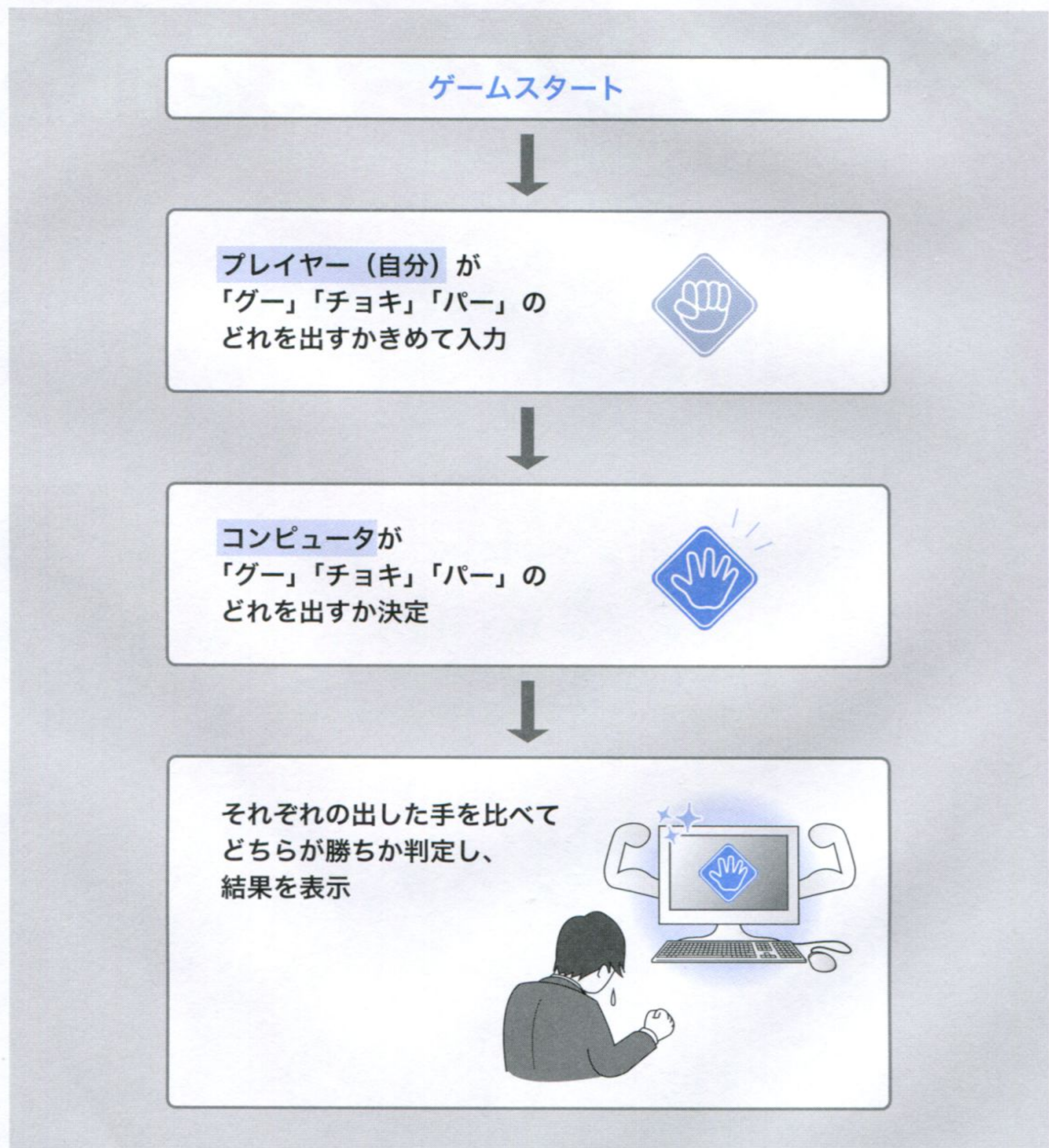
第 **2** 日

ジャンケンゲームを作ろう

いよいよ今日から本格的にＣ言語のプログラムを作ります。最初はジャンケンゲームのプログラムです。ジャンケンですから、１人では遊べません。対戦相手が必要です。コンピュータを対戦相手とするプログラムを作ります。

今日作るプログラムについて

ジャンケンゲームプログラム

コンピュータ対プレイヤーのジャンケンゲームを作ります。プレイヤーは「グー」「チョキ」「パー」に対応する数字「1」「2」「3」のどれかを入力します。一方、コンピュータの手はプログラムでランダムに決定します。お互いの手を比べてどちらが勝ったか、またはあいこかを表示します。

ゲームスタート

↓

プレイヤー（自分）が「グー」「チョキ」「パー」のどれを出すかきめて入力

↓

コンピュータが「グー」「チョキ」「パー」のどれを出すか決定

↓

それぞれの出した手を比べてどちらが勝ちか判定し、結果を表示

ゲームの実際の動作

1 ジャンケンゲームプログラムを実行すると、プレイヤーの出す手を入力する状態になる

```
C:¥source>janken.exe
【ジャンケンゲーム】
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) >
```

2 プレイヤーがグーを出す（数値の1を入力して、[Enter] キーを押す）

```
C:¥source>janken.exe
【ジャンケンゲーム】
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) > 1
```

1、2、3のどれかを入力

3 コンピュータが手を出し、勝敗が決定する

```
C:¥source>janken.exe
【ジャンケンゲーム】
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) > 1
コンピュータはチョキ！ プレイヤーの勝ち
```

4 ジャンケンは1回勝負のため、プログラムは終了

第2日 1時限目

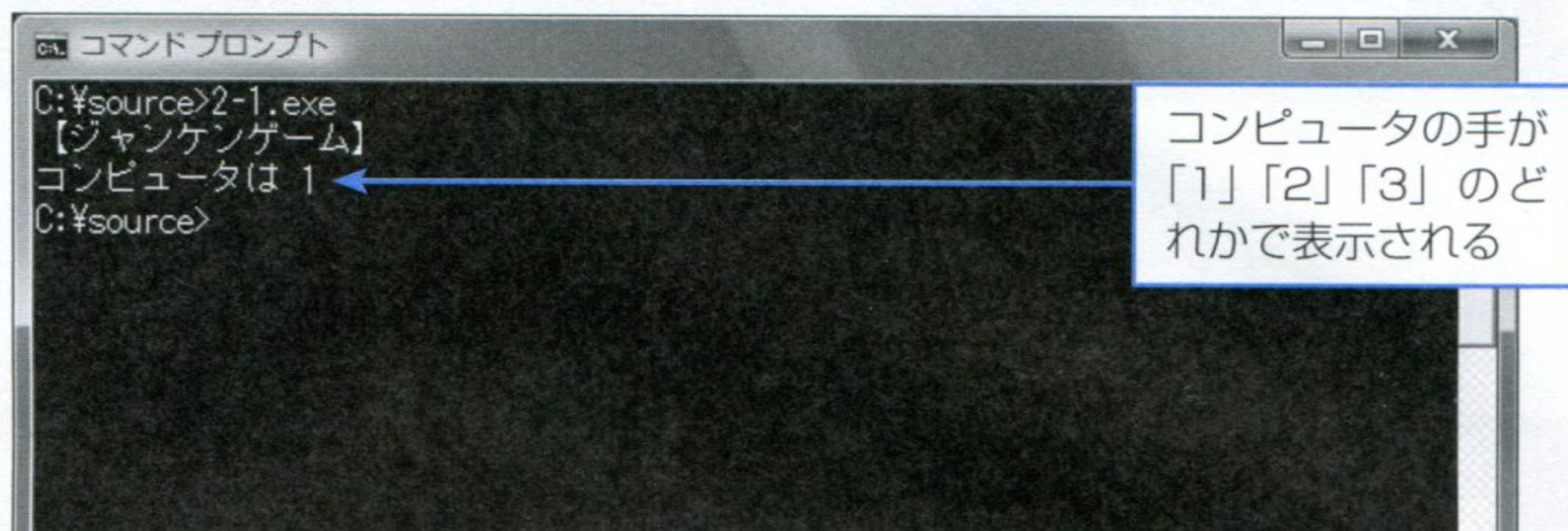
データ型について学ぼう

【ジャンケンゲームを作ろう①】

プログラム中でのデータの扱い方を学びます。データは変数という入れ物に保存して扱います。
コンピュータの出す手をランダムに決めて変数に保存し、それを表示するプログラムを作りましょう。

今回作成する例題

```
C:¥source>2-1.exe
【ジャンケンゲーム】
コンピュータは 1
C:¥source>
```

コンピュータの手が「1」「2」「3」のどれかで表示される

サンプルファイルはこちら 10days_c ▶ day02-01 ▶ 2-1.c

●このレッスンのねらい

人間同士のジャンケンでは、「ジャーンケーン、ポン！」でお互いの手を出します。互いに自分の出したい手を出しますが、相手がどの手を出してくるか、普通はわかりません。
コンピュータを相手にジャンケンをする場合も、プレイヤー（人間）は、対人間のときと同じ様に好きな手を出します。しかし、対戦相手であるコンピュータは、思考することができません。
ずっと同じ手を出すようにプログラムするのは簡単ですが、それではコンピュータが負けっぱなしになってしまいます。これではつまらないので、コンピュータが「グー」「チョキ」「パー」の中からランダムに決めて、手を出すようにしましょう。そうすれば、人間同士の勝負と同じになります。
また、ジャンケンの勝敗を判定するためには、プレイヤーとコンピュータがそれぞれどの手（「グー」「チョキ」「パー」のどれか）を出したのかを、データとして保存しておく必要があります。そのため、1時限目ではプログラム中で扱うデータについて学習します。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int computer;

    printf("【ジャンケンゲーム】¥n");
    srand(time(NULL));
    computer = rand()%3 + 1;
    printf("コンピュータは %d", computer);

    return 0;
}
```

2 入力できたら、「2-1.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント

*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、2-1.cをコンパイルする

```
C:Users¥user¥>cd ¥source

C¥source>gcc -o 2-1 2-1.c
```

4 プログラムを実行する

```
C:¥source>2-1.exe
```

▼

```
【ジャンケンゲーム】
コンピュータは 1
```

← このように「1」「2」「3」の数値のどれかが表示されれば成功！

解説

1 データの入れ物＝変数

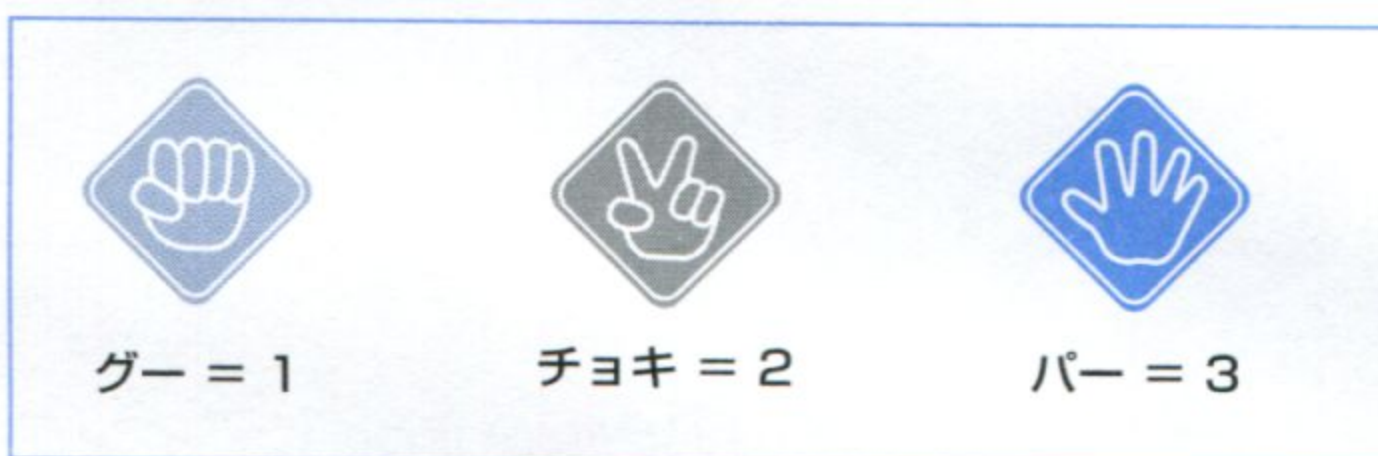
ジャンケンは、プレイヤーの出す手とコンピュータの出す手で勝敗を判定します。プレイヤーの手を扱うには「入力」が必要ですが、この部分については本日の2時限目で学習します。

この1時限目では、コンピュータの手を作り出して、データとして保存するところからはじめましょう。データの扱いは、コンピュータプログラムが行う基本のひとつです。

(1) データを保存するための変数を用意する

「グー」「チョキ」「パー」というそれぞれの手の名前をそのままデータとして持つよりも、3つをそれぞれ数値にあてはめて保存するほうが便利です。ここでは「グー」を「1」、「チョキ」を「2」、「パー」を「3」としましょう。

●ジャンケンの手を数値に置き換える

グー＝1　チョキ＝2　パー＝3

プログラムの中でデータを保存するには、データの「入れ物」が必要です。データの入れ物のことを「変数」といいます。ここでは、コンピュータの手を保存するために用意する変数の名前[*2]を、

```
computer
```

とします。この変数に、1、2、3の数値のいずれかが入ります。

(2) 変数の型を宣言し、値を代入する

1、2、3という数値は整数なので、「computer」は整数値を保存するための変数になります。

変数「computer」を使うためには、プログラムの最初の方で「この変数は、整数値を扱う変数として使います」と宣言する必要があります。これを変数宣言と呼びます。

整数値を扱う変数の宣言には、「int」というデータ型の定義を使います。コンピュータの手を「チョキ」、つまり「2」とした場合、変数の値は「2」になります。

「データ型」とは、変数に格納するデータがどういった種類のものであるかを表す識別子です。整数のデータを扱うデータ型[*3]を、「整数型」や「int型[*4]」と呼びます。

ヒント
*2：変数名は半角英数文字を使って自由につけることができます。「amari」など、ローマ字読みでの名前でもかまいません。自分でわかりやすい名前をつけましょう。なお、単純に繰り返し回数をカウントするだけの変数には、「i」や「j」などの1文字変数名がよく使われます。

ヒント
*3：「データ型」は単に「型」とも呼びます。

ヒント
*4：intとはinteger（整数）の略です。

変数の宣言は、次のように行います。

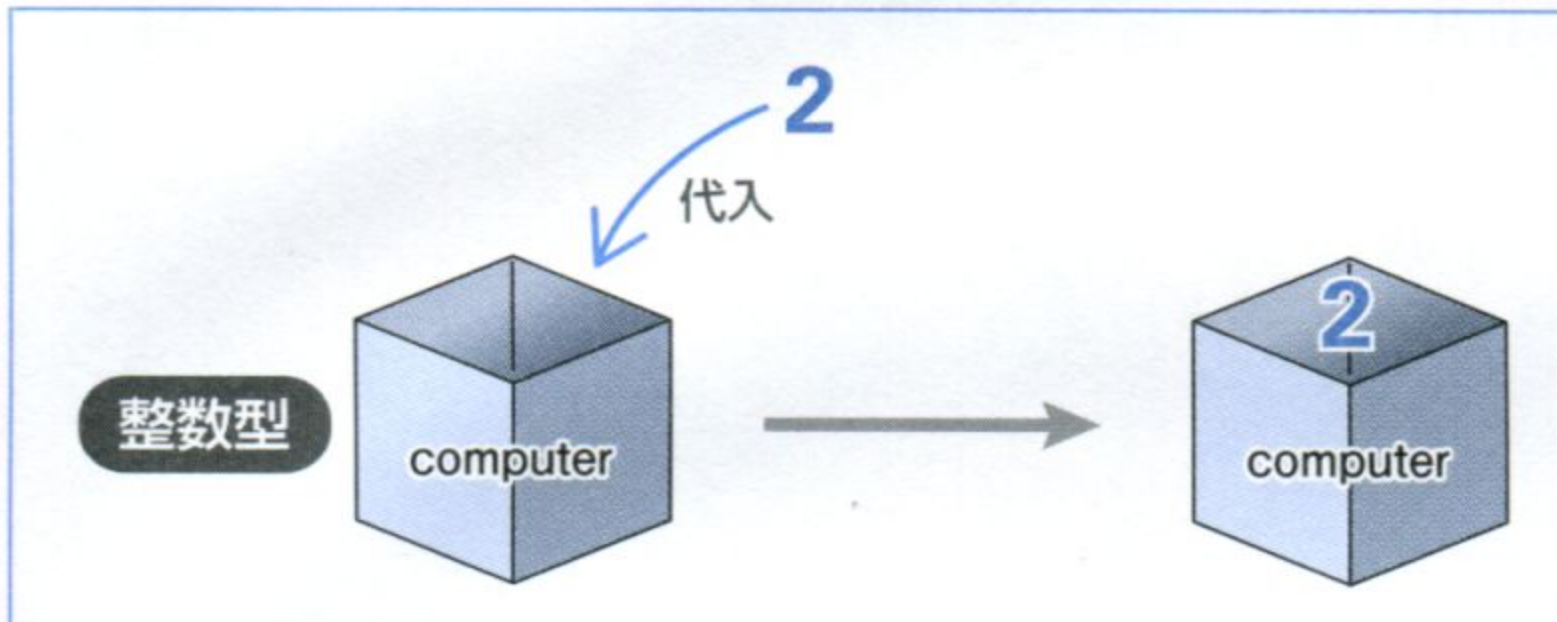
【変数宣言：宣言と同時に数値を代入する】

```
int computer = 2;
```

データ型　変数名　=　数値　;

イコール（=）記号の右側の値を左側の変数に入れることを、「代入する」といいます。つまり、この場合は、「int型の変数computerを作り、2を代入します」という意味です。

●変数に値を代入

2
代入
整数型
computer
2
computer

上の例では、変数の宣言と同時に数値を入れましたが*5、先に変数の型と変数名を宣言して、あとで数値を代入することもできます。

ヒント

*5：変数宣言と同時に値を代入することを「変数を初期化する」といいます。

【変数宣言：先に空の入れ物だけ作っておいて、あとで値を代入する】

```
int computer;
```

← int 型の変数 computer を作ります

中身は空
整数型
computer

```
computer = 2;
```

← 2 を代入します

2
代入
computer
2
computer

変数宣言をしたあと、プログラムの中でコンピュータの手を参照するには、この変数computerの中身を見ます。変数computerには、新たに数値が入れ替わるまで「2」が入っている状態になります。

COLUMN

変数を初期化しないとどうなるか？

型と変数名だけを宣言しておいて、あとで値を代入する前にその値を参照すると、予期しない値が入っていることがあります。

```
#include <stdio.h>

int main() {
    int a;
    printf("a = %d¥n", a);
    a = 2;
    printf("a = %d¥n", a);
    return 0;
}
```

このプログラムを実行してみます。「a = 2;」と値を代入する前に変数aの値に何が入っているか見ると*6、2ではない数字が入っています*7。
変数の値を参照する前には、必ず何か値を代入しましょう。

ヒント
*6：printf("a = %d¥n", a); と第2引数に変数aを指定すると、第1引数の中にある%d部分にaの値を入れて出力します。また、「¥n」は出力時に改行を指定する文字です。

ヒント
*7：プログラムの実行環境により違いがあります。

2 コンピュータの手をきめる

データの持ち方がわかったところで、ジャンケンゲームでコンピュータの出す手を実際に決定して保存するプログラムを作ってみます。

コンピュータの手をきめて画面に表示してみましょう。目標は、コンピュータの出す手が何なのか、わからないように作ることです。ここでは、「グー」「チョキ」「パー」を表す1、2、3の数字の中から、どれかひとつをランダムに決定することにします。

(1) 値を表示するプログラムを実行する

ランダムな数値を決定する前に、まず、コンピュータの手を表す変数を作り、適当な値を決め、それを表示するプログラムを作ってみましょう。

次のサンプルプログラムを作成し、コンパイルしてみましょう。

【2-1_sample1.c】

```
#include <stdio.h>

int main() {
    int computer; //コンピュータの出す手を格納する変数

    printf("【ジャンケンゲーム】¥n");
    computer = 1;                    ←変数 computer に 1 を代入
    printf("コンピュータは %d", computer);
    return 0;
}
```

```
C:¥source>gcc -o 2-1_sample1 2-1_sample1.c
```

プログラムを実行します。

```
C:¥source>2-1_sample1.exe
```

▼

```
【ジャンケンゲーム】
コンピュータは 1
```

printf関数に引数を２つ以上つけると、変数の値を出力することができます。

【printf関数】

```
printf("コンピュータは %d", computer);
```

第一引数　引数区切り　第二引数

第1引数の中にある%dの部分には、第2引数に指定した変数の値が代入されます。つまりここでは、変数computerに入っている値である1が、出力表示されます。

このような出力を「書式つき出力」と呼び、2時限目に詳しく説明します。

(2) ランダムな値を出す

2-1_sample1.cでは、コンピュータの手は、「グー」を表す数値1で固定されています。これでは、プログラムを作った人がプレイヤーになった場合には、プレイヤーが必ず勝ってしまい、おもしろくありません。コンピュータの出す手をプレイヤーが予測できないようにする必要があります。

ではどうすればよいかというと、コンピュータの手をランダムに決定すればよいのです。

ランダムな値（乱数）を決定するには、rand関数を使います。以降、本書ではrand関

数がよく出てくるので、ここでしっかりおぼえてしまいましょう。
　まず、rand関数を使うためには、

```
#include <stdlib.h>
```

をプログラムの最初に書く必要があります。この文を「stdlib.hファイルをインクルードする」といいます[*8]。printf関数を使うために、「#include <stdio.h>」を書くことと同じ意味があります。

ヒント
*8：インクルードするファイルは拡張子が.hで、このファイルを「インクルードファイル」や「ヘッダーファイル」と呼びます。

　インクルード文は、1種類のファイルにつき、プログラムの最初に1回だけ書きます。例えば、rand関数とsrand関数を使うには、両方ともstdlib.hをインクルードする必要がありますが、インクルード文を2回書く必要はありません。

```
#include <stdlib.h>
```

　これをプログラムの最初に1回書けば、rand関数とsrand関数が使えることになります。

　そして、次のようにしてrand関数を呼び出せば、乱数を求めることができます。

【rand関数】

```
rand();
```

　rand関数を呼び出すと乱数の値が返ってくるので、その返ってきた値を、用意してある変数computerに格納します。rand関数で返ってくる値は整数値です。
　それでは、rand関数を使った次のサンプルプログラムを作成し、コンパイルしてください。

【2-1_sample2.c】

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int computer;

    computer = rand();  // ランダムな整数値が代入される
    printf("コンピュータは %d", computer);
    return 0;
}
```

```
C:¥source>gcc -o 2-1_sample2 2-1_sample2.c
```

ヒント
*9：実行結果の数値は、環境により異なります。rand関数で返される値は、MinGWでは0～0x7FFFの間のどれかです。0x7FFFは16進数表現で、10進数では32767になります。この値はC:¥mingw-jp¥include¥stdlib.hに定義してあります。

プログラムを実行すると、実行結果は次のようになります。

```
C:¥source>2-1_sample2.exe
```

```
コンピュータは 130
```
*9

実行するとランダムな数値が出るはずです。ですが、この数値では意味がありません。この数値を「グー」「チョキ」「パー」を表す整数値である1、2、3に対応させる必要があります。

ランダムに作り出した値をもとに、数値1、2、3のどれかを作り出せばよいのです。それには、除算（割り算）の余りを利用します。

(3) 除算の余りを利用して乱数を加工する

一般的に、除算の結果は「商」と「余り」として算出されます。たとえば、「13 ÷ 3」では、「商」が4、「余り」が1になります。

除算は「/」で表します。13 ÷ 3は、

```
int d = 13 / 3;
```

と書き、この場合、変数dには「商」である4だけが代入されます。つまりC言語では、整数同士の除算「/」の結果は、「商」部分のみになるわけです。

13 ÷ 3の「余り」である1を算出するには、「%」を使います。13 ÷ 3の「余り」を出すには、

```
int a = 13 % 3;
```

と書きます。このとき、変数aには1が代入されます。

さて、この「余り」の性質を使うと、乱数の値と数値1、2、3の対応づけがうまくできます。

「余り＝割り切れずに残った数」なので、その値は必ず「0」から「割る数（除数）－1」までのどれかになります。たとえば割る数が3のときは、割られる数が何桁の数であっても、余りは0、1、2のどれかになるわけです。

●3で割ると、余りは0、1、2のいずれか

5 ÷ 3 = 1　余り 2
6 ÷ 3 = 2　余り 0
7 ÷ 3 = 2　余り 1
…
12 ÷ 3 = 4　余り 0

```
38 ÷ 3 = 12 余り 2
…
121 ÷ 3 = 40 余り 1
```
余りは0、1、2のどれか

割られる数が変化すれば、余りも変化します。したがって、割られる数に乱数を使い、割る数に作り出したい数の種類を使って余りを出せば、希望の範囲の数値をランダムに作り出すことができます。

```
乱数 ÷ 3 = 商 と 余り（0、1、2のどれか）

int amari = rand() % 3;
```
必ず0、1、2のどれかになる

(4) 乱数を1～3に対応づける

乱数を3で割った余りを使えば、3種類の数をランダムに出すことができました。

しかし、これでは0、1、2のどれかが算出されるので、ジャンケンの手に対応する値1、2、3に対応させるには、rand()%3の結果に1をプラスする*10必要があります。

ヒント

*10：数値の加算（足し算）には「+」記号を使います。他に、除算で使用した「/」や「%」、代入の「=」などの記号を「演算子」と呼びます。詳しくは第3日の2時限目に学習します。

```
rand()%3 + 1
```

これで0、1、2は、それぞれ1、2、3に対応できます。

ここで、今までに学習したことをまとめたプログラムを作りましょう。コンピュータの手を1、2、3のどれかからランダムに決定して出力するプログラムです。

次のサンプルプログラムを作成し、コンパイルしてください。

【2-1_sample3.c】

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int computer;

    computer = rand()%3 + 1;
    printf("コンピュータは %d", computer);
    return 0;
}
```

```
C:¥source>gcc -o 2-1_sample3 2-1_sample3.c
```

プログラムを実行します。

```
C:¥source>2-1_sample3.exe
```

```
コンピュータは 2
```
[11]

ヒント
*11：実行結果は環境により異なります。

ジャンケンの手を表す数値1、2、3のどれかが出力されるはずです。しかし、何度か実行してみるとわかると思いますが、ずっと同じ数値が出てしまいます。実は同じ環境でrand関数を使うと、毎回同じ数が出現してしまうのです。これでは意味がないので、毎回同じ数が出現しないような仕掛けを作る必要があります。

(5) 乱数の種を蒔く

それには、srand関数を使います。これは、乱数の種を蒔く関数です。

植物を育てるとき、ひまわりの種を蒔けばひまわりが育ちます。チューリップの種を蒔けばチューリップが育ちます。srand関数が行う「種を蒔く」とは、植物の種を蒔くのと同じです。同じ種を蒔けば、次にrand関数を呼び出したときにも同じ乱数ができます。異なる種を蒔けば、異なる乱数ができます。

srand関数の引数に蒔きたい「種」を指定してみます。

まず、種に数値の1を指定してみましょう。なお、このプログラムでは、乱数を3で割らずにそのまま出力しています。

次のサンプルプログラムを作成し、コンパイルしてみましょう。

【2-1_sample4.c】

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int d;

    srand(1); // 乱数の種をまく
    d = rand();
    printf("%d", d);
    return 0;
}
```
[12]

ヒント
*12：srand関数を使うためにはstdlib.hをインクルードしますが、これはrand関数を使うためにすでにインクルードしてあるので、2回行う必要はありません。

```
C:¥source>gcc -o 2-1_sample4 2-1_sample4.c
```

実行すると、適当な乱数が出力されます。

```
C:¥source>2-1_sample4.exe
```

▼

```
コンピュータは 41
```
[*13]

ヒント
*13：実行結果は環境により異なります。

もう一度実行してみても同じ値が出力されます。
次に、種の値を2に変更してみましょう。srand関数の引数を2にします。

```
srand(2);
```

上記のようにsrand関数を変更したうえで、先ほどの2-1_sample4.cをコンパイルして実行してみると、おそらく異なる値が出力されると思います。
つまり、異なる種を蒔けば異なる乱数の値が出るので、プログラムを実行するたびに異なる値を乱数の種に設定すれば、毎回異なる乱数を出すことができます。

(6) 実行するごとに違う種を蒔く

では、ここで少し考えてみてください。「プログラムを実行するたびに異なる値」とは、いったい何でしょう？ それは「時間」です。引数に現在時刻を表す数値を使えば、実行するたびに異なる乱数を発生させることができます。
引数に使う現在時刻を表す数値は、time関数を使って出します。

【time関数】

```
time(NULL)
```
[*14]

ヒント
*14：time(NULL)の値は、現在の時刻を1970年1月1日午前0時0分0秒からの通算秒数で返します。秒単位なので、仮に1秒以内に同じプログラムを実行できたら、連続して同じ乱数が出てしまいます。
time関数の引数には何もないことを意味する「NULL」を指定します。

これで、現在時刻を表す数値を取得できます。この関数の値をそのままsrand関数の引数に使えば、「プログラムを実行したときの時間」が種になるので、毎回異なる種を蒔くことができます。なお、time関数を使うには、time.hをインクルードします。

では、次のプログラムを作成し、コンパイルしてみましょう。

【2-1_sample5.c】＊2-1_sample4.cを変更

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int d;

    srand(time(NULL)); //乱数の種をまく
    d = rand();
    printf("%d", d);
    return 0;
}
```

```
C:¥source>gcc -o 2-1_sample5 2-1_sample5.c
```

プログラムを実行します。

```
C:¥source>2-1_sample5.exe
```

```
3490*15
```

この値を3で割った余りに1をプラスすれば、実行するたびに1、2、3の中からランダムな数が表示されるはずです。

3 変数にはいろいろな型がある

プログラムで何かデータを扱う場合は、変数を使用することがわかりました。

ジャンケンゲームでは、プレイヤーとコンピュータの手をそれぞれ数値として変数に保存しますが、プログラムで扱うデータは数値だけではありません。文字を扱う文字型もありますし、数値の型も整数、実数、符号の有無など、細かく分かれています。

ジャンケンゲームでは整数型[*16]のデータしか扱いませんが、他の型についてもここで説明しておきます。

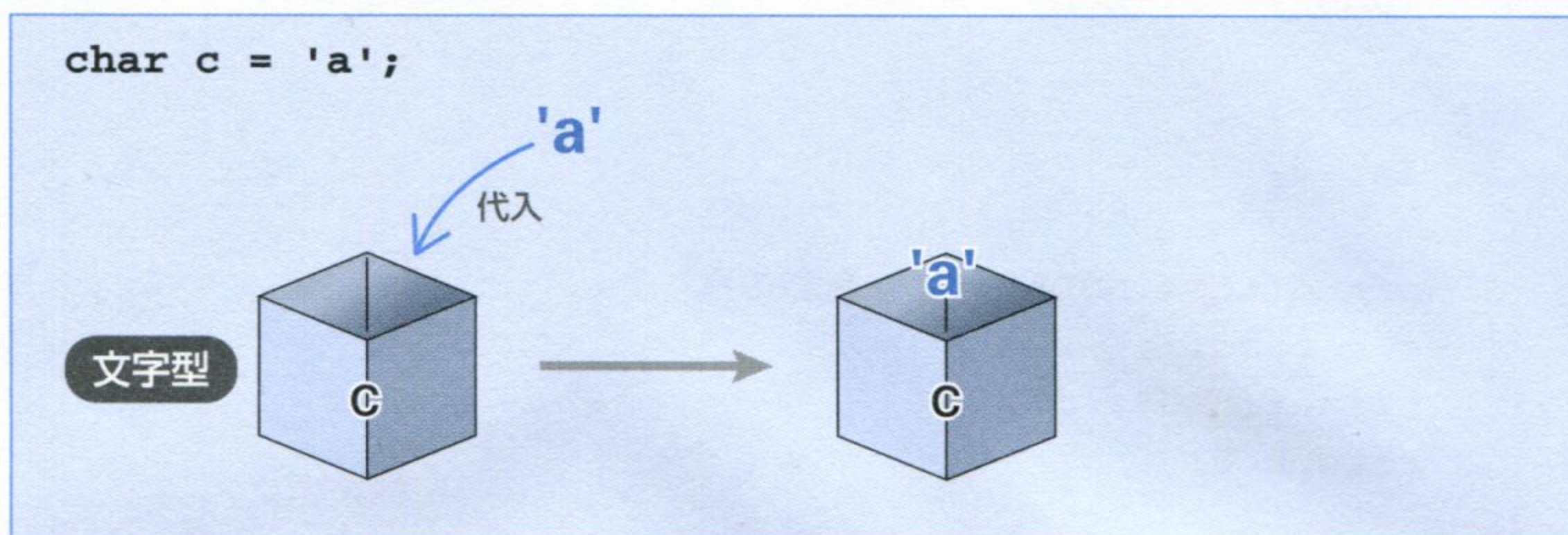
ヒント
＊15：実行結果は環境により異なります。

ヒント
＊16：整数型にはint型の他にshort型やlong型があります。しかし本書では、数値のデータ型はint型のみを使ってプログラミングを行います。他の型についての詳細は、参考としてこの時限の最後に説明します。

(1) 文字型

文字のデータ型は「char」と宣言します[*17]。数値の場合と同様、変数はプログラムの最初に宣言します。「文字」とは「a」や「b」、「_」などの記号も含めた1文字データです。

> **ヒント**
> ＊17：charはcharacter（文字）の略です。

```
char c = 'a';
```


文字型変数cに、文字「a」を代入しました。C言語では、文字を扱うときは、シングルクォート「'」で括ります。

なお次のように書くと、変数cには数値の1ではなく、「1」という「文字」が代入されます。

```
char c = '1';
```

(2) 文字列型

文字をつなげて作った文字の列は、「文字列[*18]」として扱われます。

> **ヒント**
> ＊18：正確には「文字列型」という型はありません。文字列についての詳細は、第4日に学習します。

```
char str[] = "abc";
```

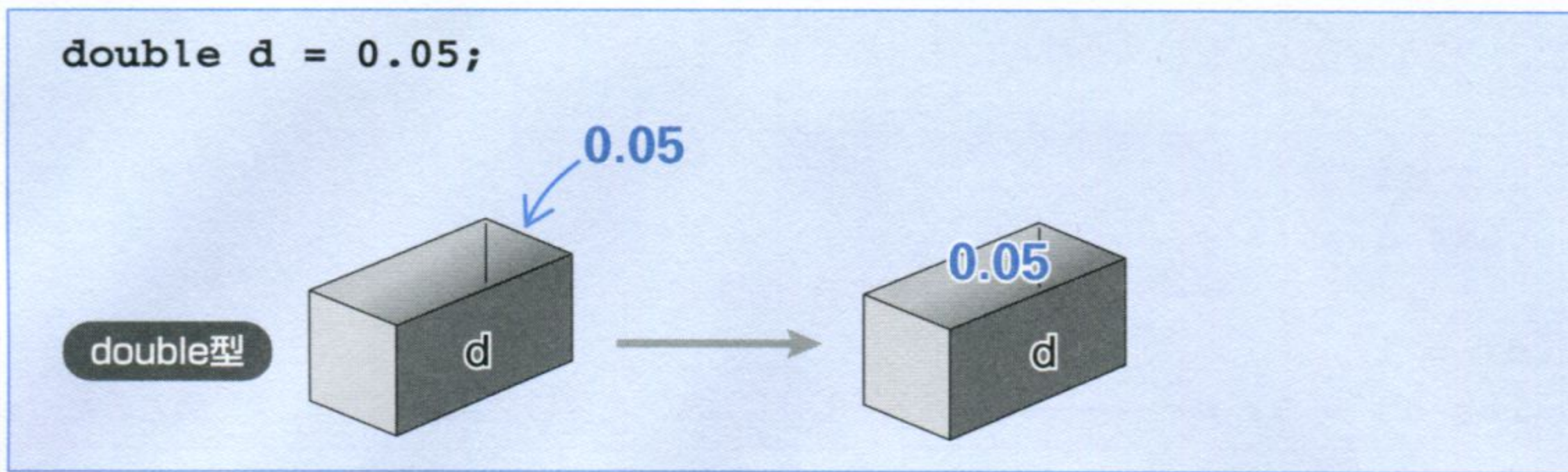
文字列変数strに文字列「abc」を代入しました。C言語では、文字列を扱うときはダブルクォート「"」で括ります。

(3) 実数型

実数はfloat型かdouble型を使います。この2つの型は、値の精度の違いによって使い分けます。double型のほうが値を保存する入れ物が大きいので、精度の高い値を扱うことができます。

次のように宣言します。

```
float f = 0.1;
```

```
double d = 0.05;
```


(4) データ型を無視して代入するとどうなるか？

変数は、宣言したデータ型の値を扱います。int型の変数には整数を代入します。値を変更するときも、同じく整数の値を代入します。

では、別の型の値を代入した場合はどうなるでしょう？

【int型変数に実数を代入してみる】

```
int d1 = 3.1;
int d2 = 3.99999;
```

どちらの変数も型がint型であるため、実数値を代入しても小数部分は切り捨てられて、実際に代入されるのは3になります。変数には、宣言したデータ型と一致する値を設定しましょう。

4 変数宣言を書く位置

プログラムの中で変数を使うためには、変数の型と変数名を宣言します。

変数の宣言は、main関数の最初に書きます。プログラム中で扱う変数が複数あるときは、すべて最初にまとめて宣言します。

同じ型の変数が複数ある場合は、

```
int d1;
int d2;
```

をまとめて、

```
int d1, d2;
```

と書くこともできます。変数の区切りにはカンマ「,」を使います。変数が初期化してあっても、まとめて書くことができます。

```
int d1 = 0, d2 = 2;
```

プログラムの途中で新しい変数が必要になった場合、途中で宣言することもできますが、コンパイラによっては途中で変数宣言を書くことはできません。プログラムの移植性を考

えて、使う変数は、なるべく最初にまとめて書きましょう。

```
int d1;
char c = 'a';
                ← ここで変数宣言がおわる
d1 = 1;
int d3 = 3;     ← ここに宣言せずになるべくプログラムの最初にまとめて書く
```

自分で作った関数[*19]の中でも、それぞれの関数内で使う変数の宣言は、その関数の最初にまとめて書きます。

ヒント
*19：自分で作る関数についての詳細は、第7日の1時限目で説明します。

まとめ

この時間では変数について学習しました。変数は、扱うデータの種類にあわせたデータ型を宣言します。

また、本書ではゲームプログラムを多く作成するので、ランダムな数値を発生させるrand関数をこれからも頻繁に利用します。ここでしっかりと理解できたでしょうか。

練習問題

Q

整数値12から17までのランダムな値を出力するプログラムを作成しなさい。使用する変数名は、自由に決めてよいこととする。

[ヒント]
6種類のランダムな値を出すようにする。

C O L U M N

データ型について

C言語の全データ型と値の範囲を、ここで紹介しておきます。
各データ型は扱える範囲がきまっています。これは、それぞれの型のデータを保存しておくために確保する、入れ物の大きさ（バイト数）が異なるからです。
例えば、整数型[*20]の変数iに0から10までの値しか代入しない場合、short型にすると、プログラムの実行時に2バイト分のメモリを使って保存されます。しかし、int型にすると、その倍の4バイト分を使うので、メモリが少々ムダになってしまいます。逆に、最高値が1,000,000,000になる変数を使いたい場合、short型では扱えません。

```
#include <stdio.h>

int main() {
    short i;

    i = 1000000000;
    printf("i:%d", i);
    return 0;
}
```

これをコンパイルすると「overflow」、つまり桁あふれのwarningが出ます。一応プログラムは実行できますが、変数iの値は正しく表示されません。
shortをintにかえてみます。すると、iの値は正しく表示されます。

```
i:1000000000
```

●C言語のデータ型と値の範囲

データ型	型の名前	バイト数[*21]	値の範囲[*21]
整数型[*22]	(signed) short	2	-32768 ～ 32767
	unsigned short	2	0 ～ 65535
	(signed) long	4	-2147483648 ～ 2147483647
	unsigned long	4	0 ～ 4294967295
	(signed) int	4	-2147483648 ～ 2147483647
	unsigned int	4	0 ～ 4294967295
文字型[*22]	(signed) char	1	-128 ～ 127
	unsigned char	1	0 ～ 255
実数型	float	4	- 3.4 × 10(指数-38) ～ 3.4 × 10(指数38)
	double	8	- 1.7 × 10(指数-308) ～ 1.7 × 10(指数308)

ヒント
*20：本書で扱う整数型のデータはすべてintになっています。プログラムがそれほど大きくない場合や、実行時の速さを極めたりするのでない場合は、特に気にせずint型を使いましょう。

ヒント
*21：バイト数と値の範囲は環境により異なる場合があります。

ヒント
*22：signedは数値にプラスマイナスの符号をつける場合に、unsignedはプラスマイナスの符号を区別しない場合につけます。singedもunsignedもつけないと、自動的にsignedと見なされます。

第2日 2時限目

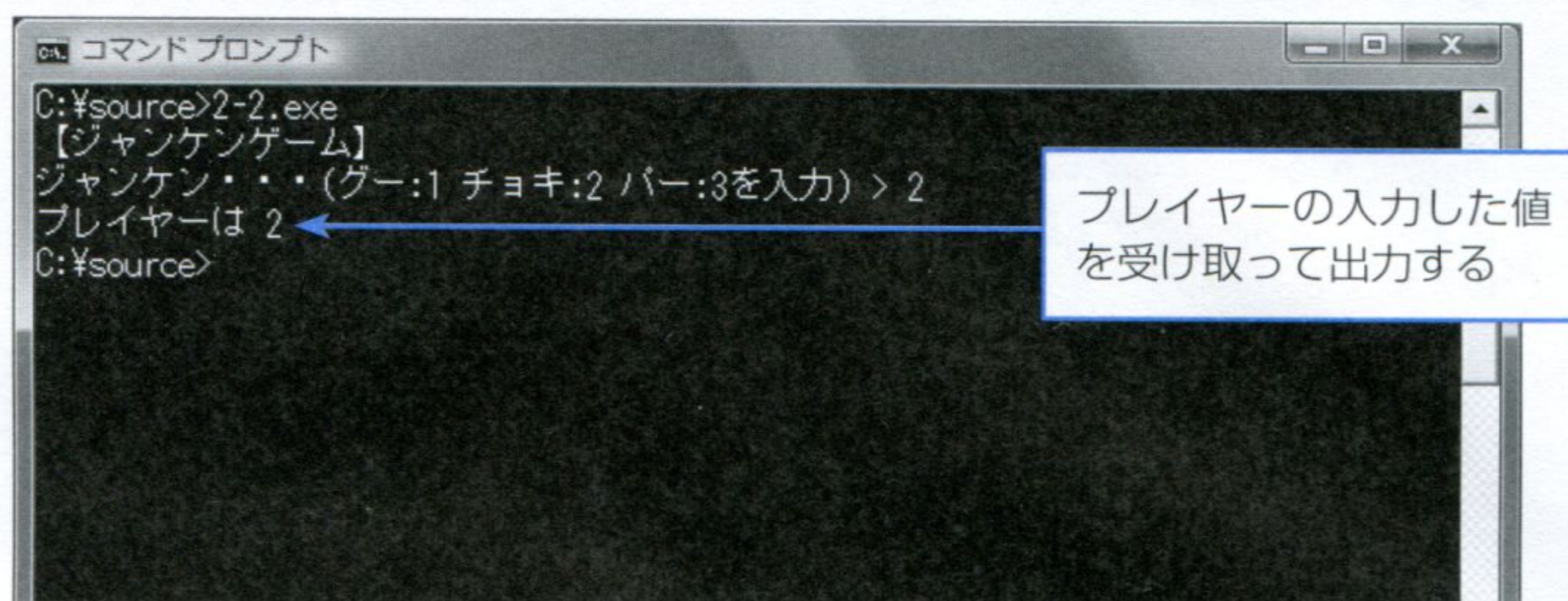
入出力のしくみを知ろう

【ジャンケンゲームを作ろう②】

1時限目ではコンピュータの出す手をきめました。次はプレイヤーの出す手を保存します。プレイヤーの手を受け取って変数に保存し、出力するプログラムを作ります。

今回作成する例題

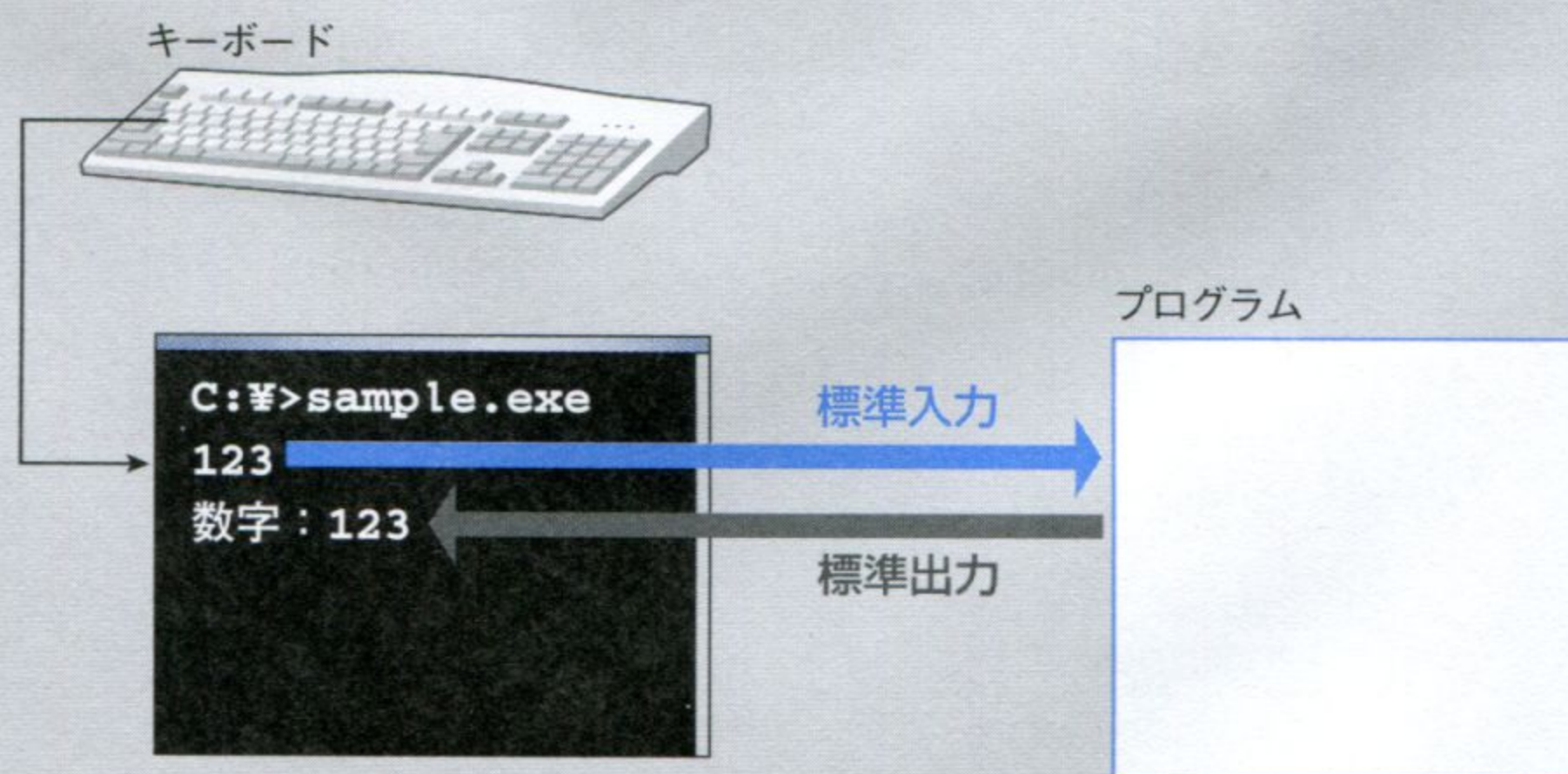
```
C:¥source>2-2.exe
【ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
プレイヤーは 2
C:¥source>
```

プレイヤーの入力した値を受け取って出力する

サンプルファイルはこちら 10days_c ▶ day02-02 ▶ 2-2.c

●このレッスンのねらい

プレイヤーは自分の好きな手をきめて、それをキーボードから入力します。プログラムがその値を受け取って、プレイヤーの手がきまります。

この、キーボードからプログラムへのデータ渡しを「入力」、または「標準入力」といいます。逆に、プログラムからコンソールへの表示を「出力」、または「標準出力」といいます。

2時限目は入力と出力について学習しましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>

int main() {
    int player; // プレイヤーの手を格納する変数

    printf("【ジャンケンゲーム】¥n");
    printf("ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > ");
    scanf("%d", &player); // 入力値を受け取る
    printf("プレイヤーは %d", player);
    return 0;
}
```

2 入力できたら、「2-2.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント
*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、2-2.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 2-2 2-2.c
```

4 プログラムを実行する

```
C:¥source>2-2.exe
```

▼

```
【ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
プレイヤーは 2
```

1、2、3のどれかを入力して［Enter］キーを押し…

その値が表示されれば成功！

解説

1 出力について

出力は、これまでに何度も出てきているので、先に説明します。

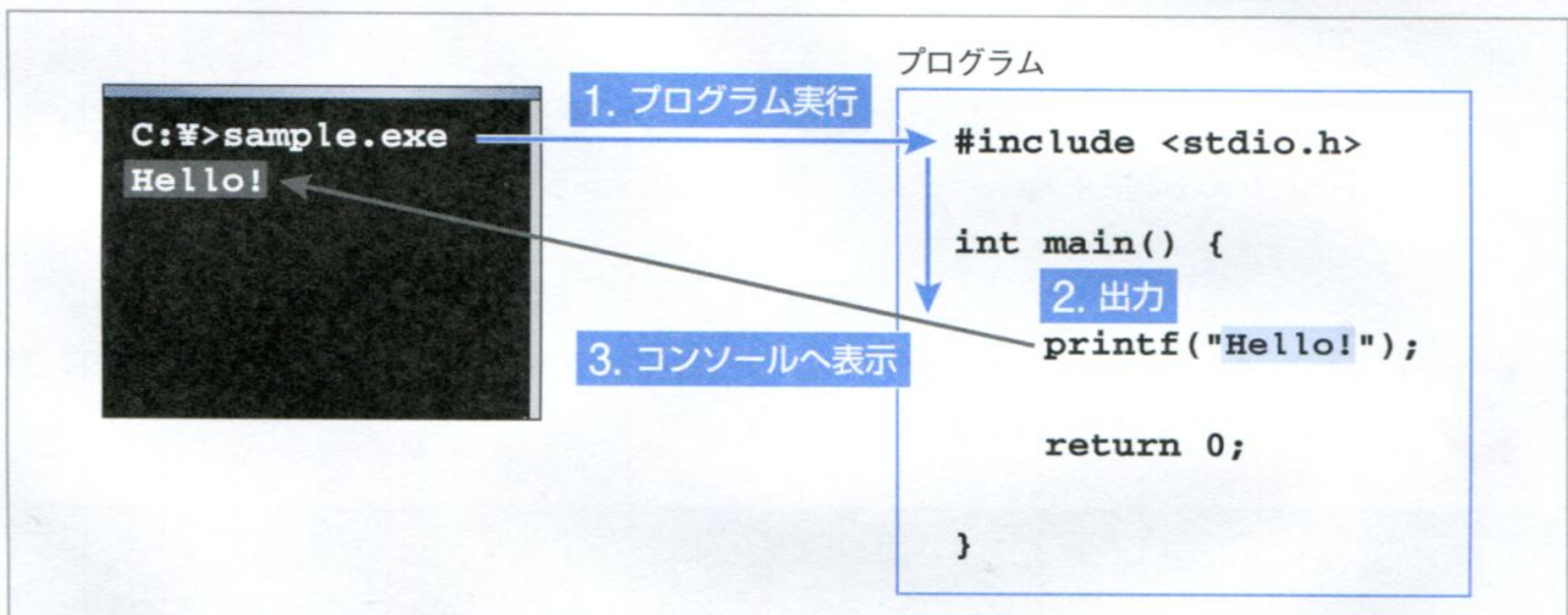
また、入力と出力をまとめて入出力と呼ぶ場合があります。キーボードからの入力とコンソールへの出力は標準入出力です。

(1) printf関数

コンソールへの出力は、printf関数*2を使います。

```
printf("Hello!");
```

これは文字列「Hello!」を出力します。printf関数で指定した引数は、そのままコンソールへ表示されます。引数の値は、必ずダブルクォート「"」で括りましょう。

●出力とprintf関数

(2) 書式つき出力

こちらもすでに何度か出てきましたが、printf関数では、書式を指定して変数の中身を出力することができます。int型変数iの値を出力したい場合は、

【2-2_sample1.c】

```
#include <stdio.h>

int main() {
    int i = 5;
    printf("数値：%d", i);
    return 0;
}
```

と指定します。printf関数の第1引数に出力書式、第2引数に書式にあてはめる変数名を指定します。このプログラムの実行結果は*3、

ヒント
*2：printf関数を使うためにはstdio.hをインクルードします。この時間に出てくる他の入出力関数もすべて同じです。

ヒント
*3：以降、本書で紹介しているサンプルプログラムのコンパイル・実行手順については、基本的に記載しません。

```
数値：5
```

となります。printf関数の第1引数「"数値：%d"」の%dの部分に、第2引数で指定した変数の値が入ります。「%d」は整数値を入れる書式です。

●書式付き出力

```
int i = 5;
printf("数値：%d", i);
```


第2引数には、変数ではなく直接値を指定することもできます。

```
printf("数値：%d", 5);
```

(3) 複数個の書式を指定する

出力したいint型変数が2つある場合は、第1引数の中に「%d」を2つ書き、第2引数に最初の%d書式にあてはめる変数名を指定し、第3引数に2つ目の%d書式にあてはめる変数名を指定します。

出力したい変数が3つになれば、同じように「%d」をひとつ増やして、引数も追加します。4つ以上でも同様に指定します。

●複数個の書式を指定

```
int a = 1, b = 2, c = 3;
printf("a: %d b: %d", a, b);
```



```
printf("a: %d b: %d c: %d", a, b, c);
```

(4) 他の書式

今まではint型の数値を出力していたため、書式は「%d」だけでした。しかし、当然、他のデータ型の値を出力するときは、それぞれにあわせた書式を指定します。

「%f」と「%lf」は実数を表し、「%c」は文字、「%s」は文字列を表します[*4]。

ヒント
*4:その他の書式については本日のおわりに紹介します。

【2-2_sample2.c】

```
#include <stdio.h>

int main() {
    float f = 0.0001;
    char c = 'a';
    char str[] = "abc";

    printf("実数f %f, 文字c %c, 文字列str %s", f, c, str);
    return 0;
}
```

```
実数f 0.000100, 文字c a, 文字列str abc
```

このとき、第1引数の中にあるカンマ「,」はダブルクォート「"」で括った中にあるので、引数区切りのカンマとは異なります。単に表示する文字としてのカンマです。

それぞれ出力するデータ型にあった書式を指定しましょう。

(5) 出力桁数を指定する

先ほどの2-2_sample2.cで実数を出力すると、0.0001ではなく0.000100と表示されてしまいました。それぞれのデータ型は表す範囲がきまっているので、最後に余計な「00」がついたのです。意味は同じですが、これをきちんと0.0001と表したい場合は、桁数の指定を行います。

桁数の指定は、書式「%f」の「%」と「f」の間に桁数の数値を入れます。実数の場合は、全体の桁数と小数部分を別々に指定するので、

【出力桁数の指定】

```
%全体の桁数.小数部の桁数f
```

と、間に小数点を表す「.」を入れて指定します。

```
float f = 0.0001;
printf("実数f%7.4f", f);
```

小数点前に全桁数、小数点後に「.」以下の桁数を指定したので、実行結果は、

```
実数f 0.0001
```

となります。小数部4桁と「.」で5桁分をとるので、残りの2桁が整数部分になります。

もしここで「%7.3f」とすると、小数部の4桁目は切り捨てられるので、「0.000」と表示されます。

なお、これらは表示上の書式指定なので、変数fの値が実際に0.000になってしまうわけではありません。また、整数部分の2桁は、この例の場合は0のみの表示なので、最初の1桁分は空白になります。

実数でできるなら、整数でも文字列でも、桁数の指定は可能です。

```
%桁数d
%桁数s
```

このように、桁数を「%」と書式指定文字の間に指定します。

```
int d = 5;
char str[] = "abc";
printf("%10d:%5s", d, str);
```

```
         5:  abc
```

スペース９個　スペース２個

int型変数dの値は5で1桁のため、「%10d」と10桁で出力指定すると、前9桁にはスペースが入ります。文字列でも同様です。残った桁数分だけ、前にスペースがつきます。この例では、文字列変数strはabcで3桁のため、「%5s」と5桁で出力指定すると、前2桁にはスペースが入ります。

また、「%-10s」と桁数にマイナスをつけると左詰めで出力します。

```
int d = 5;
char str[] = "abc";
printf("%-10d:%5s", d, str);
```

```
5_________:__abc
```
スペース９個　スペース２個

数値の前を、スペースではなく0で埋めたい場合は、桁数の前に0を指定します。

```
int d = 123;
printf("%010d", d);
```

```
0000000123
```

123は3桁なので、前7桁が0で埋まって表示されました。

(6) 制御文字の出力

printf関数の第1引数の中に「¥n」*5と書くと、コンソール上で改行されます。

実行環境によっては、プログラムの実行がすべて終了すると、自然に改行してしまうことがあります。1行だけ出力を行うプログラムでは必要ありませんが、何行かに分けて出力したい場合は、「¥n」を使うと表示が見やすくなります。

次のように、出力文字列の途中に「¥n」を入れて実行してみましょう。

```
printf("Hello¥nworld!");
```
改行を表す

```
Hello
world!
```
「Hello」のあとで改行される

文字列「Hello」と「world!」の途中に「¥n」が入っているので、その部分で改行されています。

C言語では他にも制御文字*6が存在するので、この時限のおわりにまとめて紹介します。

ヒント

*5：改行を表す「¥n」は、「¥」と「n」の2文字の組みあわせですが、実際は1文字として扱われます。
char c = '¥n';
と、char型の変数に代入できます。

ヒント

*6：制御文字のことを特殊文字とも呼びます。

2 その他の出力関数

出力用の関数として、他にfprintf関数があります。これはprintf関数と似ていますが、第1引数に出力場所、第2引数に書式、第3引数以降に出力する変数を指定する関数です。

【fprintf関数】

```
fprintf(stdout, "数値：%d", i);
```

stdout → 出力場所
"数値：%d", i → 第２引数以降は printf 関数の引数をずらしたものと同じ

上記のように、最初の引数に標準出力を表す予約語「stdout」を指定すると、次の記述と同じことになります。

```
printf("数値：%d", i);
```

またputchar関数で文字の出力を、puts関数で文字列の出力を行うことができます。

```
char c = 'a';
char str[] = "abc";

putchar(c);
puts(str);
```

3 入力について

次に、入力について説明します。ここではじめて学習しますが、入力は、ゲームプログラムでは今後もずっと必要になる機能です。

(1) scanf関数のしくみ

キーボードからの標準入力では、プログラム側でキーボードからの入力を受けつけるscanf関数を使います。

受け取った数値を格納する変数を用意しておいて、scanf関数でキーボードから入力された値を受け取り、その変数に数値を格納します。

プログラムを実行すると、scanf関数のところでとまり、キーボードからの標準入力を待ちます。入力が行われると、scanf関数がその値を受け取ってまたプログラムを再開します[*7]（次ページの上図参照）。

キーボードから何か入力し、［Enter］キーを押すと、入力が行われたことになります。

ヒント

*7：scanf関数では、何も文字を入力せず［Enter］キーを押しただけでは「入力」にならず、入力待ちの状態が続きます。

●入力とscanf関数

キーボード

3. キーボードから入力

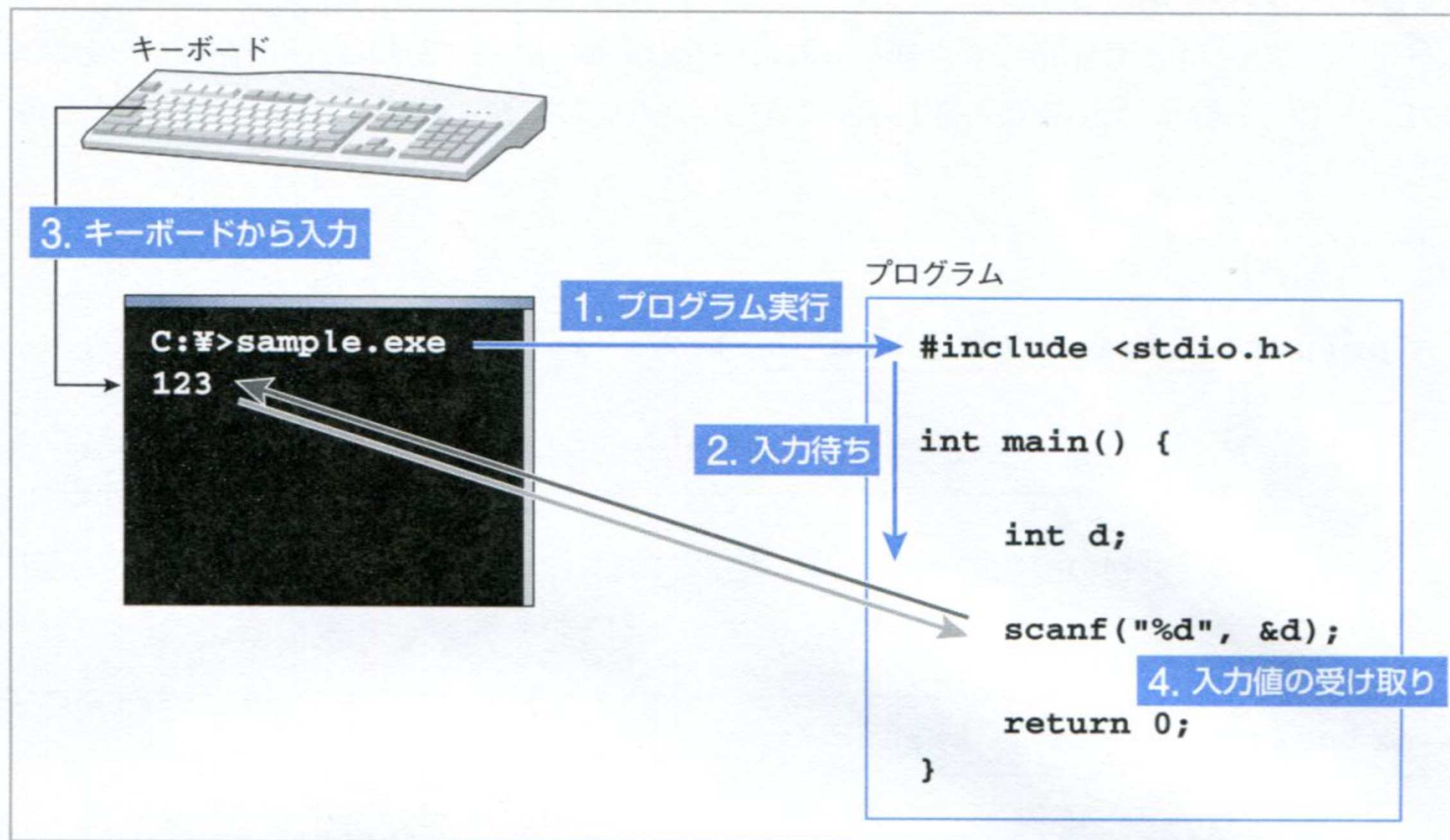
```
C:¥>sample.exe
123
```

1. プログラム実行

2. 入力待ち

プログラム

```
#include <stdio.h>

int main() {

    int d;

    scanf("%d", &d);

    return 0;
}
```

4. 入力値の受け取り

(2) プレイヤーの手を入力する

scanf関数を使ってプレイヤーの手を表す数値、「グー:1」「チョキ:2」「パー:3」を入力するプログラムを作りましょう。

まず、プレイヤーの出した手を格納する変数を

```
int player;
```

と宣言します。このint型の変数playerに、scanf関数で受け取った入力値を格納します。

【scanf関数】

```
scanf("%d", &player);
```

書式　& 変数名

●scanf関数と入力値の関係

キーボードからの入力が 1 の時

入力値 ：1

```
scanf("%d", &player);
```

数値1を受け取る　int型変数playerに格納する

scanf関数の第1引数の書式にあわせて受け取った入力値を変数playerに格納しています。「%d」は整数を表す書式です。入力して欲しい値は1、2、3のいずれかで、それを整数と

して変数playerに格納したいので、「%d」という書式を使います。

第2引数には、格納したい変数名の前に「&」をつけて指定します。これは、「変数のアドレスに格納する」という意味です。アドレスについては第5日に説明するので、今はscanf関数の第2引数の変数名の前には必ず「&」をつける、ということだけおぼえておきましょう。ただし、文字列の場合のみ、第2引数の変数名の前に「&」は必要ありません。

書式は、出力関数のprintf関数と同じです。整数型なら「%d」、文字型なら「%c」、文字列なら「%s」を指定します。

入力値を受け取って、その値を表示するプログラムが、今回課題として作ったものです。

プログラムを実行すると、scanf関数の部分でカーソルキーが点滅します。数値の入力を待っているのです。何かしら数字を入力し、[Enter] キーを押すと、scanf関数が値を受け取って、変数playerに格納します。数値を入力せずに [Enter] キーを押すことを繰り返しても、scanf関数はひたすら数値の入力を待ちつづけ、プログラムが先に進まないようになっています。

しかし、数値以外の文字（例えば「a」）や、文字列を適当に入力したらどうなるでしょう？ 結果は、

```
プレイヤーは 8855996
```

となりました。なお、この数値は環境により異なります。では、この値は何でしょう？

実は、scanf関数は指定した形式以外の入力を受け取ると、用意していた変数に値を入れてくれません。この8855996という数値は、変数playerを初期化していなかったときの値です。よって、変数playerは、本来なら0で初期化しておきましょう[*8]。

```
int player = 0;
```

今度は、数値のかわりに「3abcd」と入力してみましょう。結果は3になるはずです。scanf関数は、指定した書式以外の値が入力されると、その場で処理を終了します。このため、「3abcd」の「3」までを読み込み、あとは数値以外なので処理を終了したのです。このため、変数playerには3が入力されたことになります。

これでプレイヤーの手がきまりました。

4 その他の入力関数

scanf関数の以外の入力関数としては、gets関数とgetchar関数があります。

gets関数は、1行分の文字列を読み込みます。文字列に関してはまだ習っていないので第4日に詳しく説明しますが、ここでは、先に入力関数だけ紹介しておきます。

(1) gets関数

gets関数の引数には、受け取った文字列を格納する変数名を指定します。変数名の前に「&」をつける必要はありません。

入力した文字列をgets関数を使って変数に格納し、そのまま出力するプログラムを作ります。

ヒント

*8：変数playerの初期化処理を反映したコードは、4時限目で紹介しています。

【2-2_sample3.c】

```
#include <stdio.h>

int main() {
    char str[20]; // 半角 20-1 文字分の入力文字列を格納する変数

    gets(str); // 文字列の入力を受け取る
    printf("'%s'", str); // 入力した文字列を出力
    return 0;
}
```

```
C:¥source>2-2_sample3.exe ←プログラム実行
aaa ←文字列を入力する
'aaa' ←入力された文字列の出力
```

日本語の全角文字列[*9]も使えるので、試してみましょう。

ヒント

*9：入力する文字列は半角19文字以内にしてください。全て全角文字の場合は、9文字以内で入力してください。

(2) getchar関数

getchar関数を使ってみましょう。getcharはキーボードからの入力を1文字だけ受け取る関数なので、引数は必要ありません。getchar関数で受け取った文字を、格納する変数に代入します。

【2-2_sample4.c】

```
#include <stdio.h>

int main() {
    char c; // 入力文字を格納する変数

    c = getchar(); //1 文字受け取って代入する
    printf("%c", c); // 入力した文字を出力
    return 0;
}
```

```
C:¥source>2-2_sample4.exe  ←プログラム実行
a        ←入力文字
a        ←先程入力した文字を出力する
C:¥source>2-2_sample4.exe
abc      ←入力
a        ←出力
```

getchar関数は1文字だけ受け取って出力するので、文字列「abc」を入力すると、このプログラムでは最初の1文字だけ受け取って、出力しています。

では、2文字目以降の部分はどうなってしまうのでしょう？ 実は、プログラム中ではまだ残っています。読み込んだ文字列は、バッファと呼ばれる領域に格納されるのです。

getchar関数はそこから1文字ずつ取ってくるので、続けてgetchar関数を実行すると、その次の文字を取得することができます。

●getchar関数と入力値の関係

キーボード

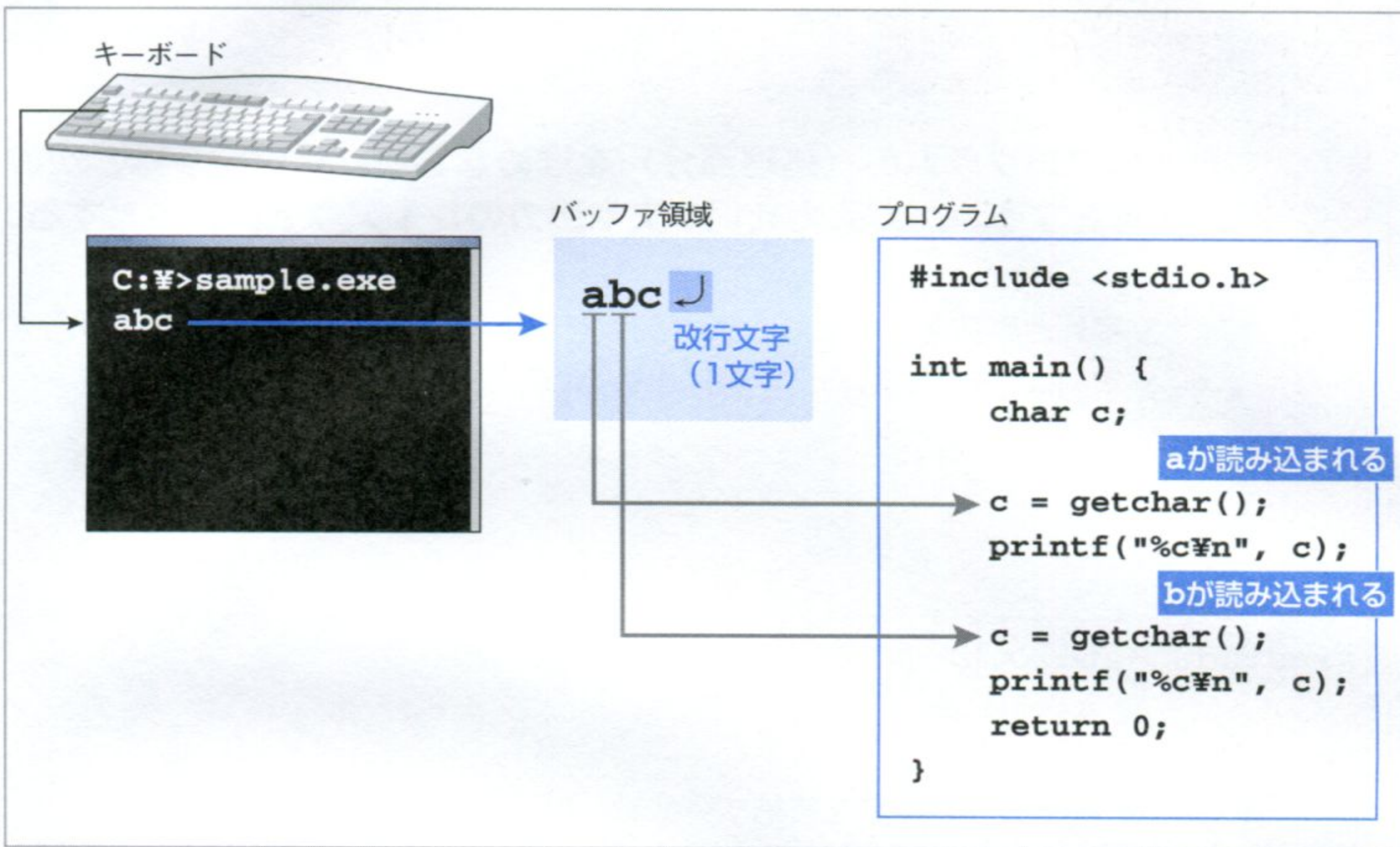
```
C:¥>sample.exe
abc
```

バッファ領域

abc

改行文字
（1文字）

プログラム

```
#include <stdio.h>

int main() {
    char c;
                        aが読み込まれる
    c = getchar();
    printf("%c¥n", c);
                        bが読み込まれる
    c = getchar();
    printf("%c¥n", c);
    return 0;
}
```

```
abc  ←―― 入力文字列
a    ←―― 出力
b    ←―― 出力
```

文字列「abc」と入力した場合、最後に押した［Enter］キーも改行文字としてバッファと呼ばれる領域に入ります。読み出しは、そのバッファから行われます。計4文字を読み出すには、getchar関数を4回呼ぶ必要があります。

最後の改行文字まで読み出さずに他の入力関数を使うと、バッファに文字が残っているので予期しない動きをすることがあります。

ここでは入力関数を3種類紹介しました。入力にどの方法を使うかは、プログラムの目的により使い分けましょう。

まとめ

入出力は、今度、どのプログラムにも出てきます。
入出力関数では、データの種類によって正しい書式を指定しましょう。
特に入力では、入力内容とプログラムの書き方によっては予期しない動きをすることがあるので、よく注意してプログラムを作成する必要があります。

練習問題

Q

次のプログラムの（処理部分）を埋めなさい。処理内容は、次の内容とする。なお、入力を促す文や出力のレイアウトは自由とする。

・文字列と実数をそれぞれscanf関数を使って入力し、その値を出力する。
・文字列は右寄せ１０桁、実数は小数点以下３桁で出力する。

```
#include <stdio.h>

int main() {
    double d;    // 実数を格納する変数
    char str[10]; //10-1 文字以内の文字列を格納する変数

    （処理部分）

    return 0;
}
```

解答は巻末に

COLUMN

制御文字

printf関数の中で出力文字列中に「¥n」と書くと、改行が出力されます。このように、出力文字列で特別な意味を持つ組みあわせを制御文字といいます。
ここでは、C言語で使用できる制御文字を紹介します。制御文字は、どれも「¥」と他の１文字の組みあわせになっていますが、その２文字をあわせて１文字として扱われます。

●C言語の制御文字

制御文字	意味
¥a	ベルを鳴らす
¥b	１文字戻る
¥f	改ページ
¥n	改行する
¥r	復帰（同じ行の左端に移動）
¥t	水平タブ
¥v	垂直タブ
¥¥	¥自身
¥'	シングルクォート(')
¥"	ダブルクォート(")
¥0	ヌル文字

例えば、

```
printf("¥a");
```

はベルが、つまりビープ音が鳴ります。
それぞれどのような効果があるのか、試してみましょう。
なお、「¥」はバックスラッシュを表します。スラッシュ「/」とは向きが逆なのでバックスラッシュと呼びますが、日本語キーボードでは、この「\」が存在しません。このため、かわりに「¥」を使っています。

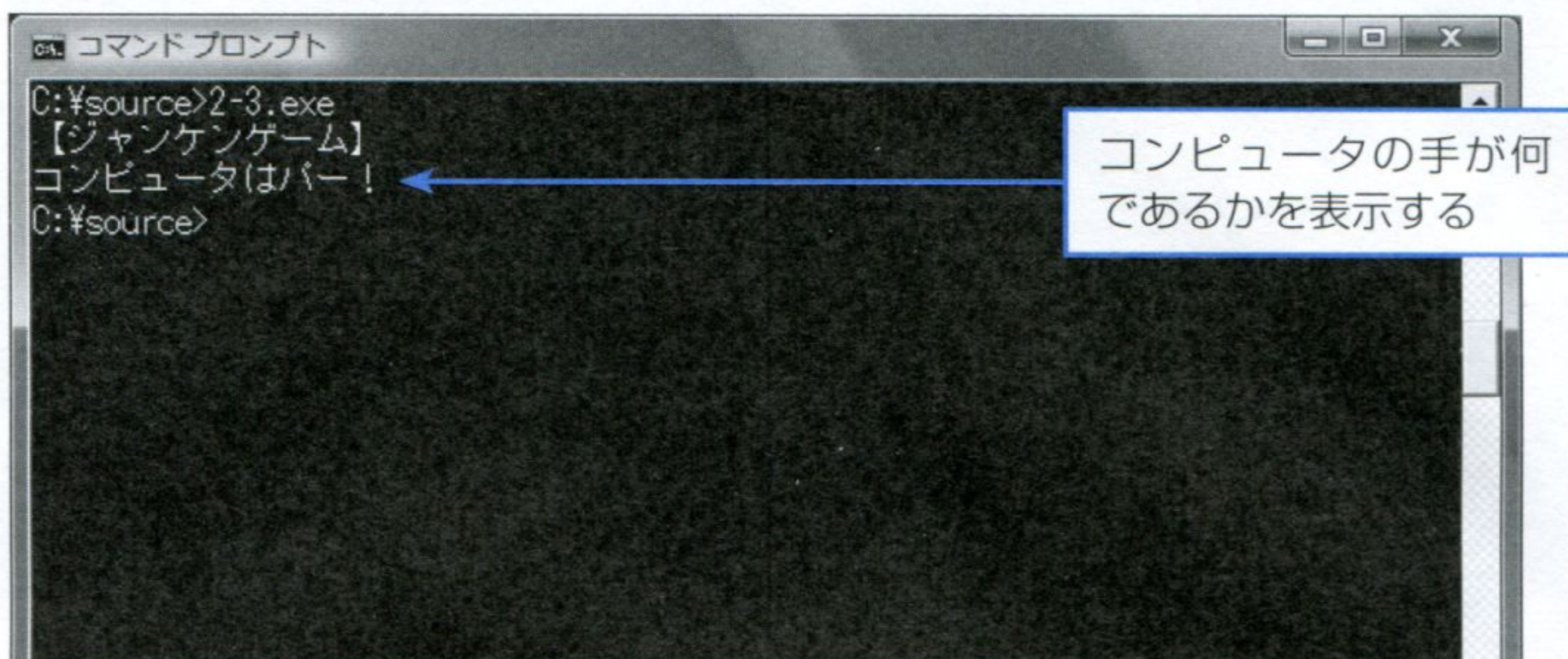
第2日 3時限目

【ジャンケンゲームを作ろう③】分岐処理を理解しよう

ジャンケンゲームでは、プレイヤーの入力した手とコンピュータの手を比較して勝敗を判定します。本レッスンでは、コンピュータの手をわかりやすく表示するプログラムを作ってみます。

今回作成する例題

```
C:¥source>2-3.exe
【ジャンケンゲーム】
コンピュータはパー！
C:¥source>
```

コンピュータの手が何であるかを表示する

サンプルファイルはこちら 10days_c ▶ day02-03 ▶ 2-3.c

●このレッスンのねらい

1時限目では、「グー」「チョキ」「パー」に対応する数値1、2、3を、数値のまま出力していました。しかし、数値だけが表示されてもプレイヤーにはわかりにくいので、きちんと「グー」「チョキ」「パー」と表示してあげましょう。
コンピュータの手が1の場合は、「グー」と表示します。1ではなく2の場合は、「チョキ」と表示します。3の場合は、「パー」と表示します。
つまり、コンピュータの手の値により、処理が分かれます。このように処理が分かれることを、分岐処理といいます。本レッスンでは、分岐処理の扱いをマスターします。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int computer;

    srand(time(NULL));
    printf("【ジャンケンゲーム】¥n");
    computer = rand()%3 + 1;
    printf("コンピュータは");
    if(computer == 1) { printf("グー"); }
    else if(computer == 2) { printf("チョキ"); }
    else { printf("パー"); }
    printf("！");
    return 0;
}
```

2 入力できたら、「2-3.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント
*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、2-3.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 2-3 2-3.c
```

4 プログラムを実行する。実行するたびに「グー」「チョキ」「パー」のどれかが表示されれば成功！

```
C:¥source>2-3.exe
```

```
【ジャンケンゲーム】
コンピュータはパー！
```

```
【ジャンケンゲーム】
コンピュータはグー！
```

```
【ジャンケンゲーム】
コンピュータはチョキ！
```

解説

1 分岐処理とは

条件によって行う処理を分ける方法を、分岐処理と呼びます。

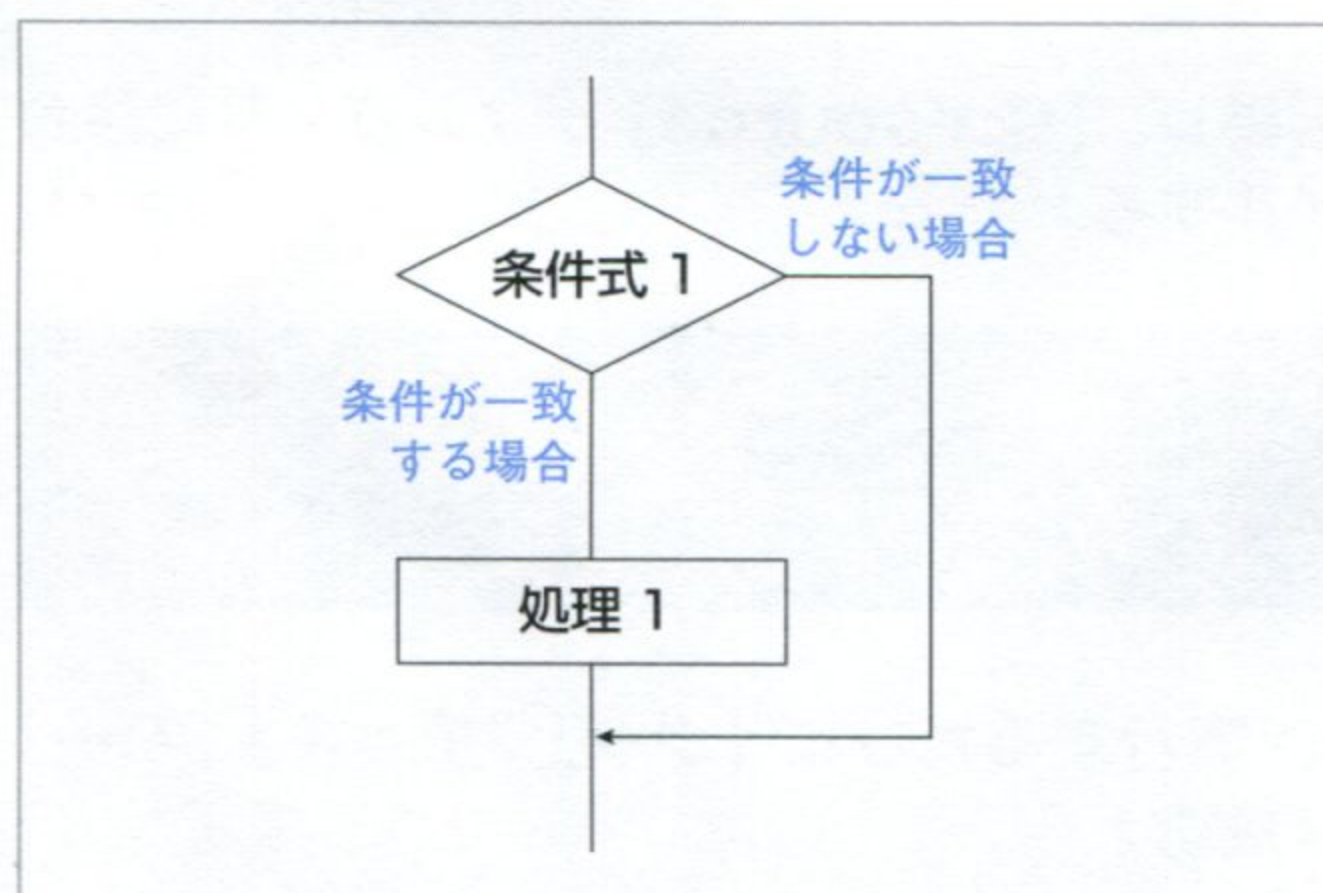
分岐処理の書き方は非常に簡単です。

(1) if文

if文は、もしも条件1に一致するなら処理1を行う、という書き方をします。条件1に一致しないなら、処理1は行いません。

【if文】

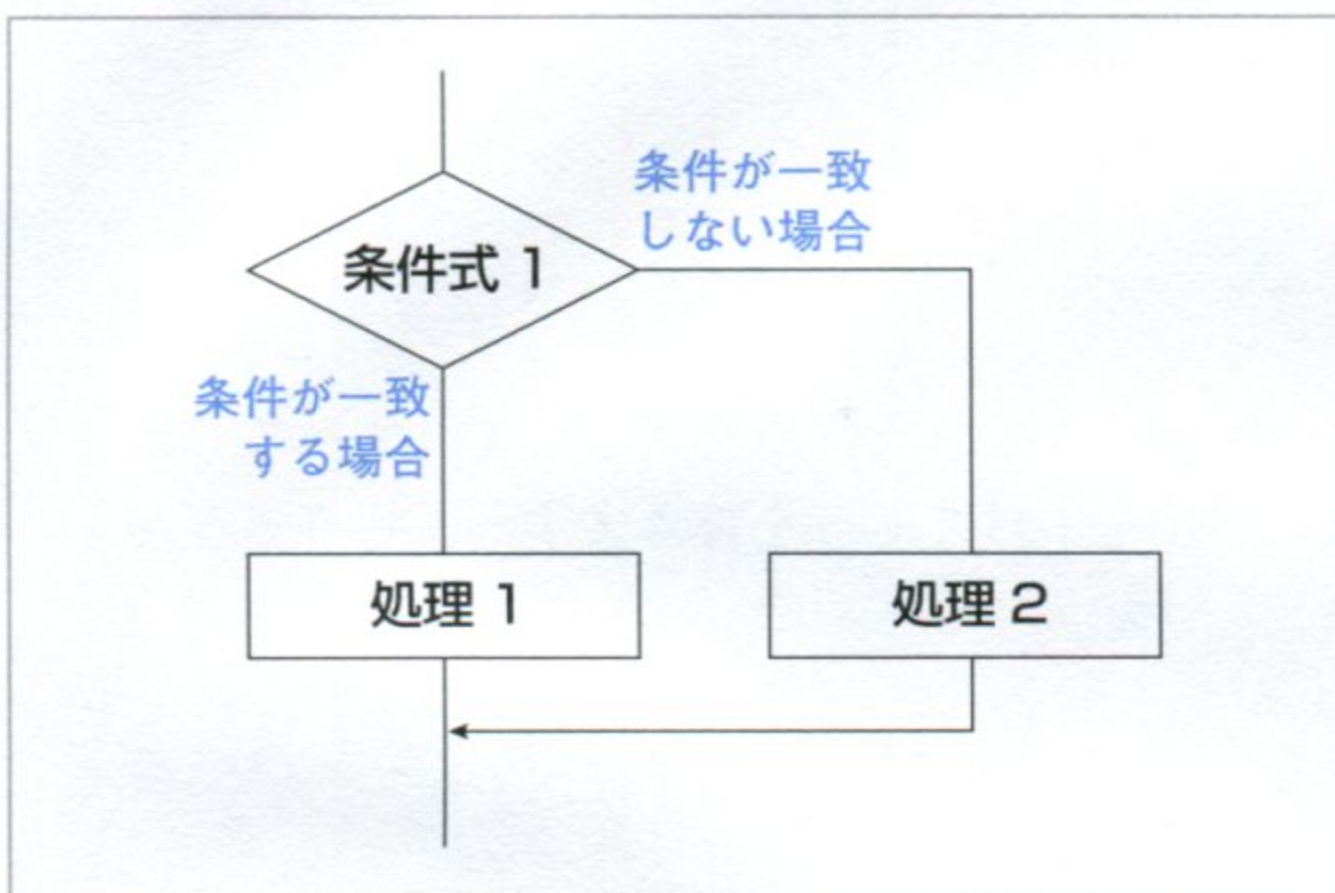
```
if( 条件 1) {
    処理 1;
}
```

●if文の考え方

(2) if～else文

if～else文では、条件に一致しない場合の処理も指定します。もしも条件1に一致するなら処理1を行い、条件1に一致しないなら処理2を行います。

【if～else文】

```
if( 条件1) {
    処理1;
} else {
    処理2;
}
```

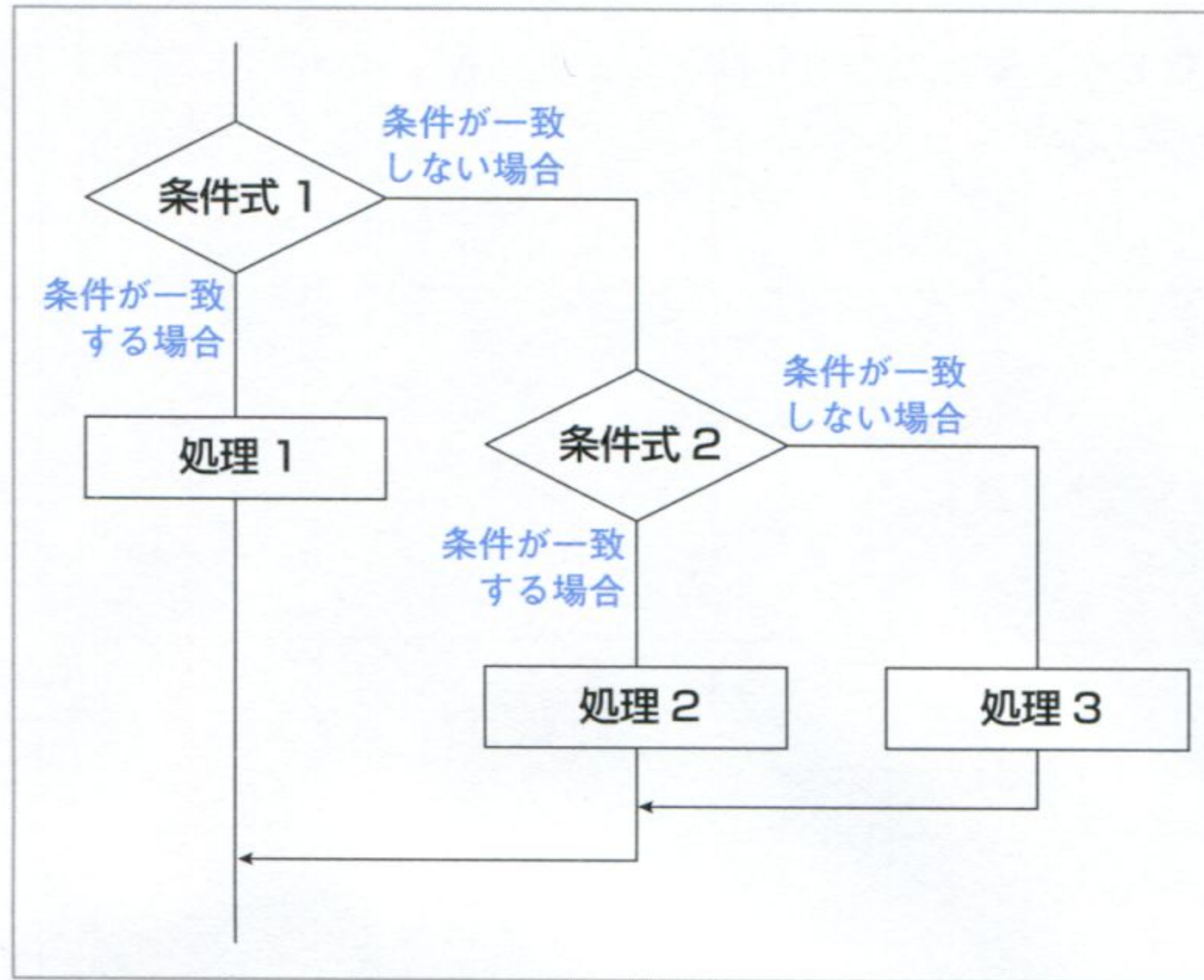
●if～else文の考え方

(3) if～else if文

最初の条件に一致しない場合の処理で、さらに条件をつけて分岐を行いたい場合は、if～else if文で次のように書きます。

【if～else if文】

```
if( 条件1) {
    処理1;
} else if( 条件2) {
    処理2;
} else {
    処理3;
}
```

●if～else if文の考え方

条件式 1
条件が一致しない場合
条件が一致する場合
処理 1
条件式 2
条件が一致しない場合
条件が一致する場合
処理 2
処理 3

このように、elseのあとに続けて次の条件であるif(条件2)を書きます。すべての条件にあてはまらなかった場合として、最後にelseをつけて処理を書きますが、行う処理がない場合、この部分は必要ありません。

また、else ifはいくつでも増やすことができます。

2 条件式とは？

では、条件に一致しているのか、それとも一致していないのかは、どのように判断するのでしょう？

（1）真と偽

ジャンケンの手を表示する場合、コンピュータの手が1ならば「グー」と表示する処理を行います。これをif文の形にすると、次のようになります。

```
if( 手が 1 である) {
    「グー」と表示する処理 ;
}
```

表示する処理にはprintf関数を使います。ここではじめて出てくるのが、「条件式」です。

「手が1である」という条件式は、「コンピュータの手 == 1」と書きます。「=」記号を2つ並べると、イコールの意味になります。なお、「=」をうっかりひとつだけしか書かないと、ただの「代入」になってしまうので、注意しましょう。

上のif文をコードにすると、次のようになります。

```
if(computer == 1) {
    printf("グー");
}
```

条件式が正しかったり条件式と一致したりする場合を、「真」といいます。その逆を「偽」といいます。C言語では0を偽とし、それ以外（特に1）を真とします。

分岐処理は、真偽を用いると次のように理解できます。

【分岐処理】

```
if(条件式) {
    条件式が真の時の処理;
} else {
    条件式が偽の時の処理;
}
```

今回の場合、「コンピュータの手が1であれば真、1でなければ偽」となります。

(2) 条件式が逆の場合

「==」を使った条件式では、「==」の左右にある変数や数値が一致する場合に真になります。

```
if(左項目 == 右項目) {
```

では、逆に「2つの項目が一致しない」というのが条件式であった場合を考えてみましょう。条件式と分岐処理の書き方は、次のとおりです。

【条件式に!を用いた分岐処理】

```
if(左項目 != 右項目) {
    条件式が真の時の処理;
} else {
    条件式が偽の時の処理;
}
```

「!=」で「一致しない」という意味になります。「!=」の左右にある変数や数値が一致しない場合に、真になります。「一致しない」が真、つまり正しいということになるので、慣れないと妙な感じがしますが、こういう条件を判定しなければならない場合も出てくるので、上手に使いこなせるようになりましょう。

当然、行うべき処理についても注意しなければなりません。例えば、次の分岐処理を逆に書いてみましょう。

```
if(computer == 1) {
    printf("グー");
} else {
    printf("グー以外");
}
```

▼

```
if(computer != 1) {
    printf("グー以外");
} else {
    printf("グー");
}
```

一致しない場合の処理が真になるので、if～elseの処理の内容が入れかわります。

「==」や「!=」は、その左右にある項目を比較して真偽を判定するので、比較演算子といいます。

この他の比較演算子には、「○○は△△より大きい・小さい」を表す「>」や「<」などがあります。これらの演算子は、「==」や「!=」と同じく条件式に使うことができます。他の演算子については、第3日の演算子のところで改めて紹介します。

(3) 比較演算子を使わない条件式

if文の条件式には、比較演算子を使わない場合もあります。先ほど、「C言語では0を偽とし、それ以外（特に1）を真とする」と説明しました。つまり、比較演算子を使わなくても、条件式にあたる部分が真偽のどちらになるのかを判定できればよいのです。

一番簡単な例は、

```
if(1) { 処理; }
```

です。条件式が無条件で1、つまり真なので、この部分は必ず処理を行います。

これを変数を使って応用すると、

```
int flag = 1;
if(flag) { 処理; }
```

となります。変数flagが0ならば、処理は行いません。

3 ジャンケンの手の表示

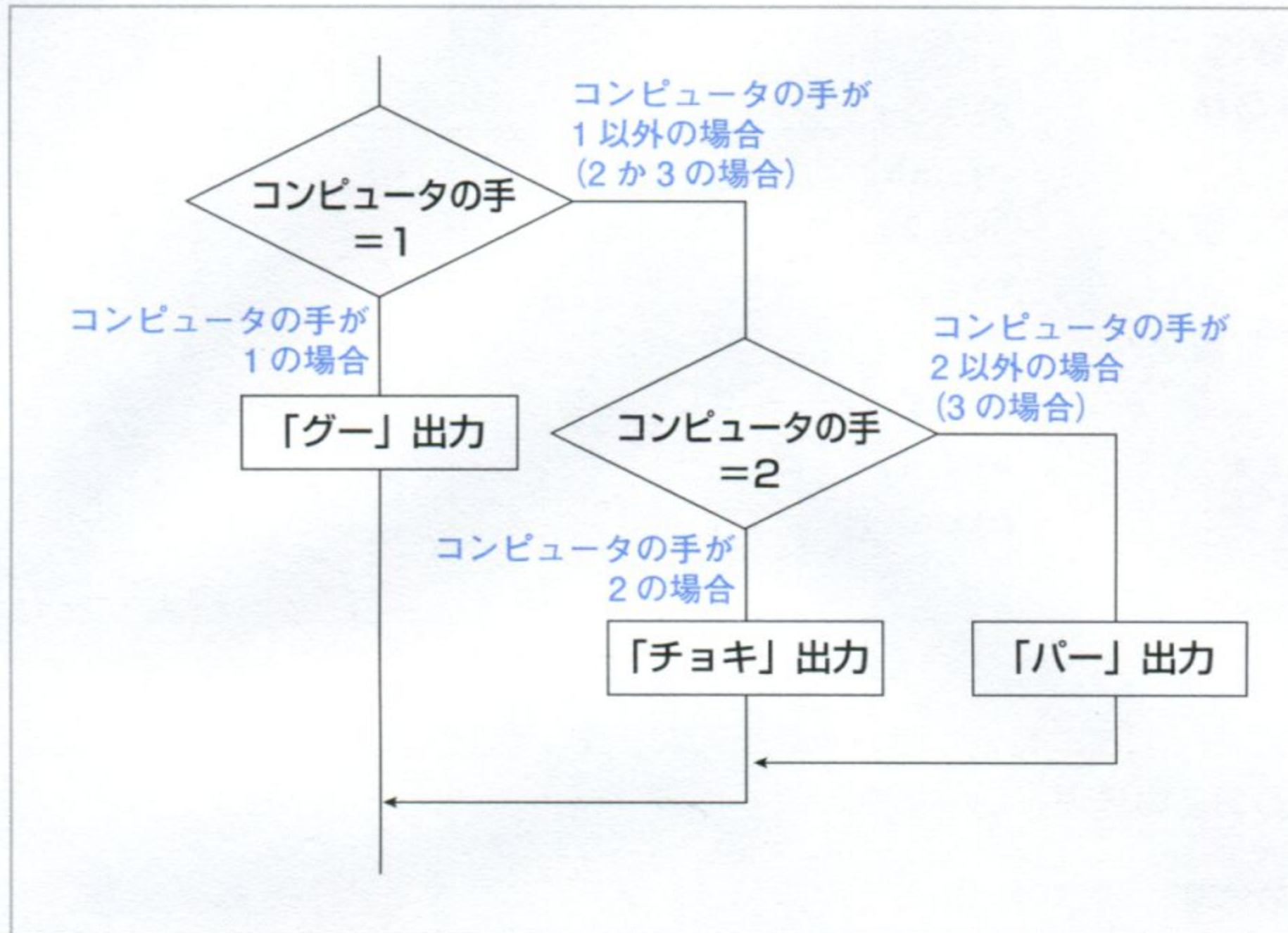
ifを使った分岐処理の書き方がわかったところで、ジャンケンゲームでコンピュータの手を表示する部分のプログラムを作ってみましょう。

コンピュータの手が1であれば、「グー」と表示します。1以外のときは、コンピュータの手は2か3、ということになります。1以外の場合でコンピュータの手が2であれば、

「チョキ」と表示します。

コンピュータの手はrand関数を使って作り出しているので、「1」「2」以外の残りは、必ず3になります。よって残りの場合は、無条件で「パー」と表示します。

この分岐処理を実現するためには、「if～else if～else」文を使います。

●グー、チョキ、パーを表示する処理

最初にグーの場合の処理、違ったらチョキであるか判定して、そこであてはまったらチョキの処理、どちらにもあてはまらなかったらパーの処理をします。

冒頭で作成したプログラムの、該当部分を見てみましょう。

```
if(computer == 1) { printf("グー"); }
else if(computer == 2) { printf("チョキ"); }
else { printf("パー"); }
```

最初の条件式である「computer == 1」は、「変数computerが数値の1と一致するなら」という意味です。

最後のパーの処理は、

```
else if(computer == 3) { printf("パー"); }
```

としたほうがよいのですが、コンピュータの手は乱数で取得していて、このプログラムの場合は1か2でなければ必ず3のはずなので、elseだけにしています。正確に書きたい人は「else if」を利用してください。

4 switch文による分岐処理

C言語では、if文の他にもswitch文を使った分岐処理の書き方があります。

if文もswitch文も条件による分岐ですが、switch文はその名のとおり、スイッチのように条件をピンポイントで指定します。

【switch文：整数値を判定】

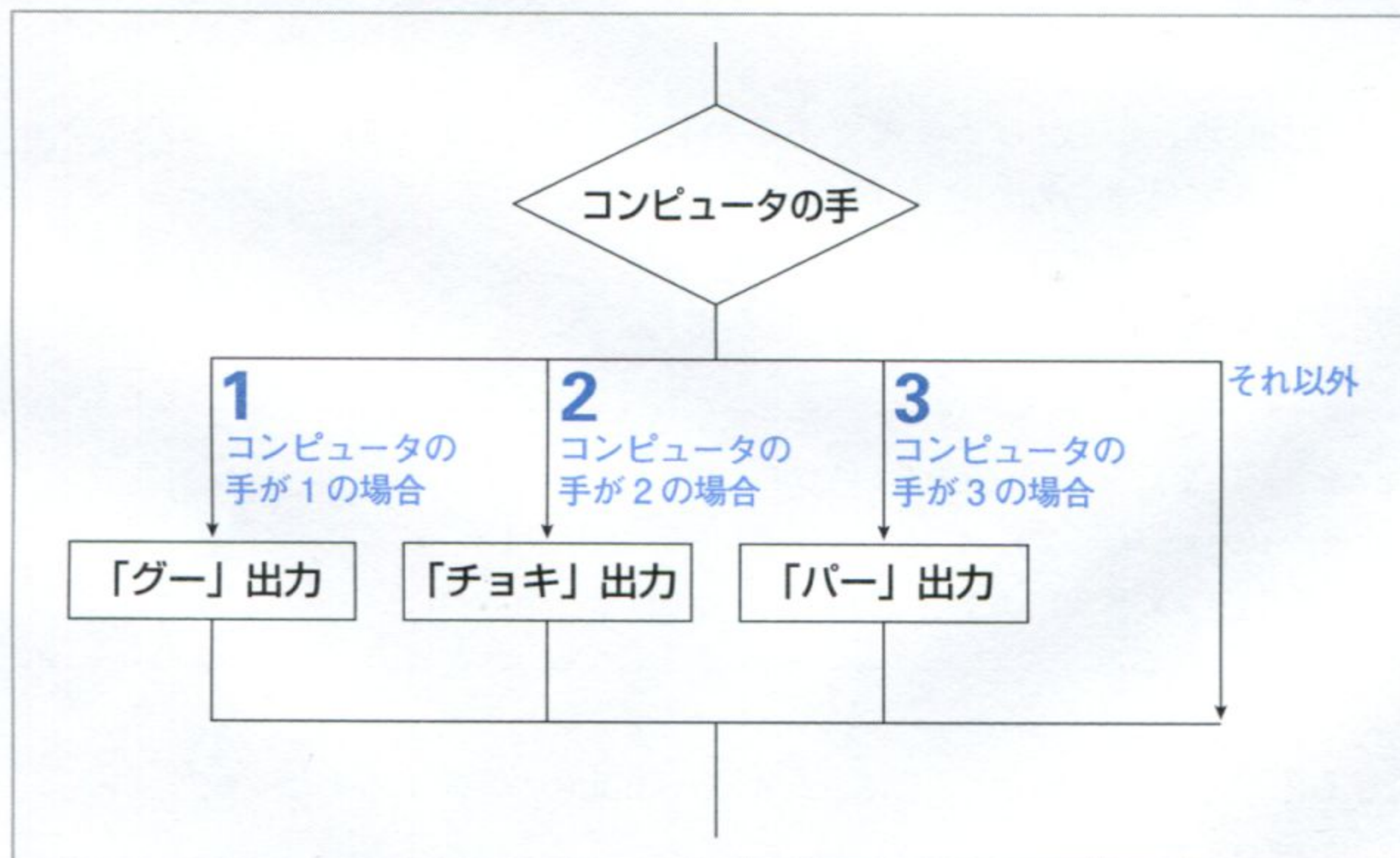
```
switch ( 変数名 ) {
    case 変数値1:     処理1;
                      break;
    case 変数値2:     処理2;
                      break;
    case 変数値3:     処理3;
                      break;
    default:          処理4;
                      break;
}
```

ヒント

*2：変数値のあとは、コロン「:」です。セミコロンと間違えないように注意しましょう。

switchに続く括弧「( )」に、判定したい変数を入れます。変数値*2がcaseのあとの値にあてはまれば、そのあとの処理を行います。

●switch文による分岐処理

コンピュータの手
1 コンピュータの手が1の場合
2 コンピュータの手が2の場合
3 コンピュータの手が3の場合
それ以外
「グー」出力
「チョキ」出力
「パー」出力

各caseの最後に「break;」という文を入れています。処理の中でこれが来ると、次からのcaseを判定せずに、switch文は終了します。「break;」を入れなければ、次のcase文も次々と見ていきます。

最後の「default:」は、「どのcaseにもあてはまらない場合」です。if～else文でいうところの、elseと同じ扱いになります。「default:」で何もすることがない場合は、「break;」のみで終了します。

if～else文と同様に、分岐の数だけ、caseは増やすことができます。ひとつのcaseに複数の処理を書くこともできます。ブロックは必要ありません。

なお、switchで判定できる値は、整数値か文字だけです。文字の場合の書き方は、次のとおりです。

【switch文：文字を判定】

```
char c = 'a';
switch (c) {
    case 'a':    処理1;
                 break;
    case 'b':    処理2;
                 break;
    default:     break;
}
```

if～elseとswitchのどちらを使うかは自由ですが、条件式がすべて「変数==数値」または「変数==文字」で、さらに分岐が多い場合は、switch文を使った方が見やすくなります。

今回の分岐は3つだけなので、if～else文を使いましょう。

まとめ

処理の分岐にはif～else文かswitch文を使います。
ifの分岐には条件式を使用し、結果は真か偽で判定します。

練習問題

Q

今回の例題のプログラムにあるif～else部分を、switch文を使って書き換えなさい。

解答は巻末に

第2日 4時限目

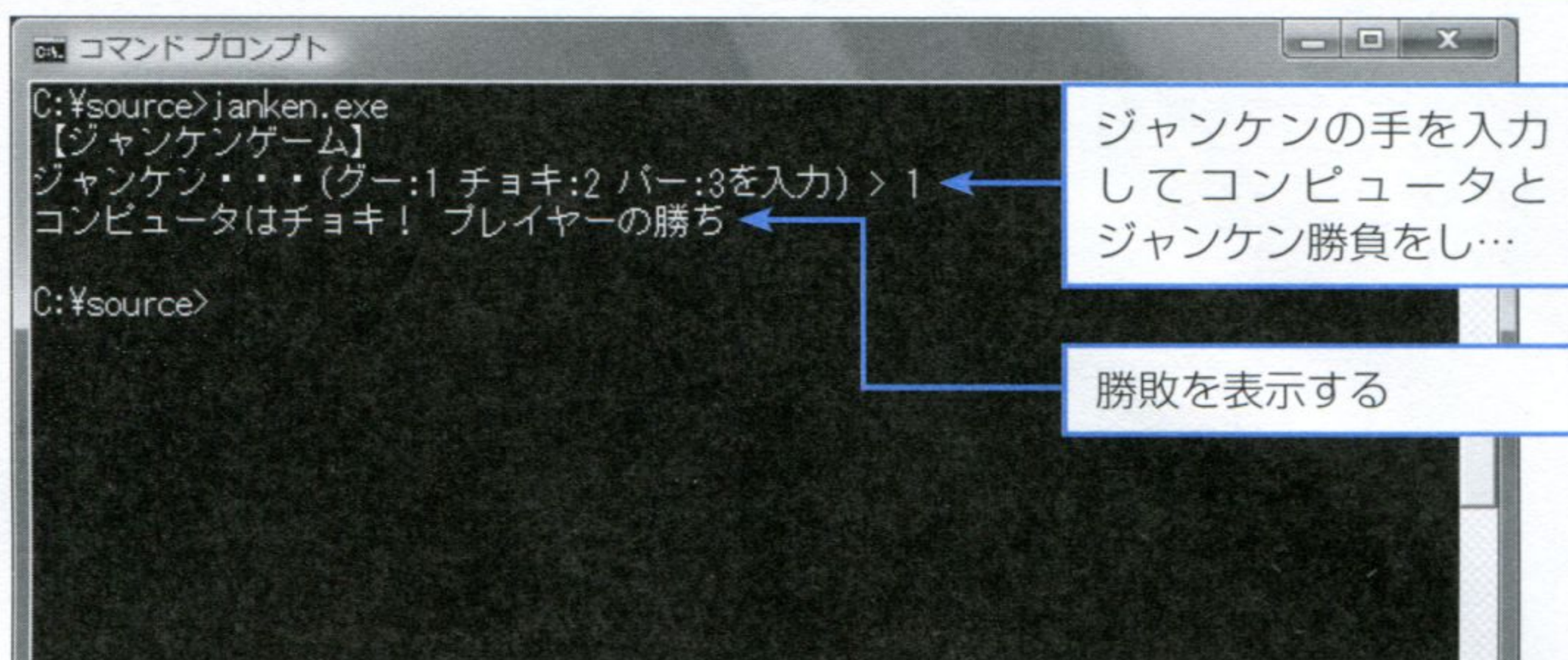
【ジャンケンゲームを作ろう④】ジャンケンゲームを完成させよう

ジャンケンゲームを完成させましょう。

今回作成する例題

```
C:¥source>janken.exe
 【ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ！ プレイヤーの勝ち

C:¥source>
```

ジャンケンの手を入力してコンピュータとジャンケン勝負をし…

勝敗を表示する

サンプルファイルはこちら 10days_c ▶ day02-04 ▶ janken.c

●このレッスンのねらい

本日の１時限目～３時限目までで、プレイヤーの手とコンピュータの手を出すプログラムをそれぞれ作成しました。これらのプログラムを組みあわせ、最後にコンピュータの手とプレイヤーの手を比較して勝敗を判定し、ジャンケンゲームを完成させます。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

/* ジャンケンゲーム（繰り返しなし）*/
int main() {
    int player = 0, computer;

    printf("【ジャンケンゲーム】¥n");
    // 乱数の種をまく
    srand(time(NULL));

    // プレイヤーの手の入力
    printf("ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > ");
    scanf("%d", &player);

    // コンピュータの手の入力
    computer = rand()%3 + 1;
    printf("コンピュータは");
    if(computer == 1) { printf("グー"); }
    else if(computer == 2) { printf("チョキ"); }
    else { printf("パー"); }
    printf("！ ");

    // 勝敗の判定と結果表示
    if(computer == player) {
        printf("あいこ¥n");
    } else if(player == 1) {
        if(computer == 2) { printf("プレイヤーの勝ち¥n"); }
        else { printf("コンピュータの勝ち¥n"); }
    } else if(player == 2) {
        if(computer == 3) {  printf("プレイヤーの勝ち¥n"); }
        else { printf("コンピュータの勝ち¥n"); }
    } else if(player == 3) {
        if(computer == 1) { printf("プレイヤーの勝ち¥n"); }
        else { printf("コンピュータの勝ち¥n"); }
    }
    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「janken.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、janken.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o janken janken.c
```

4 プログラムを実行する。勝敗の判定結果が正確に表示されれば成功！

```
C:¥source>janken.exe
```

▼

```
【ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ! プレイヤーの勝ち
```

```
【ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはパー! あいこ
```

解説

1 勝敗の判定方法

勝敗の判定方法を説明します。

まず、普通のジャンケンの判定方法を考えてみましょう。両者の手が同じ場合は、「あいこ」です。それ以外の判定は、次のとおりです。

●ジャンケンゲームの勝敗

・プレイヤーの手が１（グー）
　コンピュータの手が２（チョキ）の場合 →プレイヤーの勝ち
　コンピュータの手が３（パー）の場合 →コンピュータの勝ち
・プレイヤーの手が２（チョキ）
　コンピュータの手が３（パー）の場合 →プレイヤーの勝ち
　コンピュータの手が１（グー）の場合 →コンピュータの勝ち
・プレイヤーの手が３（パー）
　コンピュータの手が１（グー）の場合 →プレイヤーの勝ち
　コンピュータの手が２（チョキ）の場合 →コンピュータの勝ち

これをif〜else if文にそのままあてはめれば、勝敗の判定が完成します。

```
if(computer == player) {
    printf("あいこ¥n");                    入れ子
} else if(player == 1) {
    if(computer == 2) { printf("プレイヤーの勝ち¥n"); }
    else { printf("コンピュータの勝ち¥n"); }
} else if(player == 2) {
    if(computer == 3) { printf("プレイヤーの勝ち¥n"); }
    else { printf("コンピュータの勝ち¥n"); }
} else if(player == 3) {
    if(computer == 1) { printf("プレイヤーの勝ち¥n"); }
    else { printf("コンピュータの勝ち¥n"); }
}
```

else if文の中に、さらにif〜else文が入っています。このような状態を「入れ子」と呼びます。

実は、この判定の書き方はあまりスマートではありません。よりスマートに書くには、「複数の条件を一度に判定できる方法」を使う必要があります。この方法については、次の第3日で学習します。

2 プログラムの書き方

これまでに作成したそれぞれのパーツを組みあわせて、プログラムを作ります。

ここまでいくつかのプログラムを実際に書いてみて、C言語のプログラムの書き方はだいたいわかってきたと思いますが、ここで改めて復習しておきましょう。

(1) 必要なファイルをインクルードする

まず、必要なファイルをインクルードします。どんなプログラムでもprintf関数ぐらいは使うでしょうから、C言語のプログラムを書くときは、

```
#include <stdio.h>
```

を最初に入力するようにしましょう。

他のインクルード文も、わかっている時は最初から書いてしまいましょう。例えばランダムな数値を出すプログラムの場合は、次の2つが必要になります。

```
#include <time.h>
#include <stdlib.h>
```

もしくは、プログラム中でインクルード文が必要になる関数が出てきたら、そのたびに追加しましょう[*2]。

ヒント

*2：必要ないファイルをインクルードしてもエラーにはなりませんが、これはムダな作業です。プログラム中で特定の関数を使うのに必要なものだけを、インクルードしましょう。

(2) 空のmain関数と正常終了の証を書く

C言語のプログラムでは、main関数が必ず必要です。インクルード文の次は、中身が空のmain関数と、main関数の最後に正常終了の証である「return 0;」文を書いておきましょう。

```
int main() {

    return 0;
}
```

(3) main関数のブロックに実際の処理を書く

あとは、main関数ブロックの中に処理を書いていけばよいだけです。

ブロックの中でもきまりごとが存在しました。使用する変数の宣言は、なるべく処理の前にまとめて書きましょう。

```
#include インクルードファイル名

int main() {
    変数宣言 ; // コメント

    処理 ;
    return 0;
}
```

ブロックを表す括弧「{}」は、必ず対になっている必要があります。プログラムが長くなると、うっかりブロック終了の}を書き忘れてしまうことがあるので、できたら最初から{}を対で書いておきましょう。

または、次のように{と}が縦に同じ位置にくるような書き方をすると、コードが見やすくなります。

```
int main()
{
    if( )
    {
        :
    }
}
```

あとは処理の機能ごとに適度に空行を入れて見やすくし、コメントもできるだけ入れるようにします。例えば変数の宣言では、変数名を定義したあとに続けて「この変数は何である」とコメントを書いておくとよいでしょう。

3 ジャンケンゲームプログラムを書く

プログラムの書き方に従って、ジャンケンゲームを完成させましょう。

まず、stdio.hとtime.h、stdlib.hをインクルードします。stdio.hはprintf関数を使うため、time.hはtime関数を使うため、stdlib.hはsrand関数やrand関数を使うためにインクルードします。

次にmain関数を作り、中身を書いていきます。main関数の最初に、

player：プレイヤーの手
computer：コンピュータの手

この2つの整数型の変数を宣言します。

```
int player = 0;
int computer;
```

1行にまとめて書いてもかまいません。そして、変数playerだけは0で初期化しておきます*3。

続いて乱数の種を蒔いておきます。srand関数はプログラムの中で最初に1回だけ行えばいいので、最初に書いておきます。

あとは、単純に1時限目～3時限目で作成したプログラムを組みあわせます。今回はコンピュータの手をきめるより先にプレイヤーの手を決定したいので、2時限目で作った部分を先に持ってきます。最後に、この4時限目で作った勝敗判定部分をつければ終了です。

また、このジャンケンゲームではプレイヤーが手を入力する部分に、「1はグー、2はチョキ、3はパーです。どれかを入力してください」という意味の入力プロンプトを表示しています。こうした部分のレイアウトは、自由に作成してください。

```
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) >
```

入力プロンプト　　ここにプレイヤーが手を入力

ヒント

*3：プレイヤーの入力では、きちんと整数が入力されるとは限らないため、変数playerの初期化を行います。

さて、このプログラムを「janken.c」というファイル名で保存し、コンパイルします。コンパイルが成功したらjanken.exeを実行してみます。

勝敗判定が正確に行われていることを確認しましょう。

まとめ

今回作ったジャンケンゲームで、C言語の基本中の基本であるデータの扱いと標準入出力、および分岐処理について学習しました。どれも今後のゲームプログラムを作るのに必要となる知識です。しっかり理解してから、次の学習に進みましょう。

練習問題

Q

作成したジャンケンゲームプログラムを、プレイヤーが間違った手を出したとき（1、2、3以外の値が入力されたとき）にメッセージを出すように改造しなさい。

解答は巻末に

C O L U M N

バイトとは

データ型の大きさを表すのに、「バイト」という言葉を使いました。理解している人も多いと思うので、ここでは簡単に説明します。
コンピュータ上で扱うデータは、すべて「ある」「なし」という2つのデータで成り立っています。これを1と0に対応させると、コンピュータ上のデータは、すべて0と1で扱われることになります。0と1で成り立つデータ、それは2進数のデータとして置き換えることができます。2進数に置き換えれば、あとは10進数にも対応できます。
2進数とは、基数が2である数値表現です。通常、私たちは数値を10進数で数えています。10進数の場合、0、1、2、……9で桁がひとつあがります。10を基本にして桁がひとつ繰りあがるしくみです。2進数とは、その基本となる数を2にしただけです。つまり、0、1で繰りあがります。したがって、10進数の2は、2進数では10と書きます。0、1で次の桁にあがってしまうので、2進数は常に0と1だけで表現されます。

●10進数と2進数の対応

10進数	2進数
1	1
2	10
3	11
4	100
5	101

この2進数で表されるひとつ分のデータを、「ビット」といいます。例えば、101（2進数表現）と3桁ある場合は、3ビットのデータといいます。

3ビットのデータの場合、表現できるデータの種類は

000	001	010	011	101	101	110	111

の8種類です。これを10進数に置き換えると、0から7までの数値を表現できます。
同じく4ビットでは、0000から1111まで、つまりは10進数の0から15までを表現できます。
このように、ビット数が増えれば表現できるデータの範囲が広がります。
ビットをもうすこし増やし、8ビットをひとかたまりとした単位を「バイト」と呼びます。1バイトでは、0から255まで扱えます。short型は2バイト、つまり16ビットを使えるので、0から65535までの数値を表現できるのです。
また、コンピュータ上で文字「a」は「01100001」と表せます*4。これは2進数8桁で十分足りているので、文字型は1バイトです。

ヒント

*4：文字とそれを表す数値の対応を文字コードといいます。これについては第4日に説明します。

C O L U M N

書式について

printf関数やscanf関数で使用するデータ型の書式指定一覧を示します。

●データ型の書式

書式	意味
%d	符号付き整数（int）
%u	符号なし整数（int）
%ld	整数（long）
%f	符号付き実数（float）
%lf	符号付き実数（double）
%c	文字
%s	文字列
%x	符号なし整数で16進数表示
%o	符号なし整数で8進数表示
%p	ポインタ

ここで、16進数と8進数表示について簡単に説明します。どちらも2進数と同じく基数を16か8にした数値の表現方法です。
8進数は0、1、2、3、……、7で1桁あがり、10、11、12、……と表現します。こちらは簡単です。
同様に考えると、16進数は0、1、2、3、……、9、10、11、……、15で、1桁あがることになります。しかし、これではすでに次の桁にあがっています。
よって、0、1、2、……、9の次は、10を表す数字のかわりに「A」と書きます。同様に11はB、12はC、15はFと書きます[*5]。つまり、16進数表現の10は、10進数での16にあたります。

10進数の27を、%d、%x、%oでそれぞれ出力してみます。

```
int i = 27;
printf("%d %X %o", i, i, i);
```

```
27 1B 33
```

10進数での27は、16進数では1B、8進数では33と表現されることがわかりました。

ヒント
*5：A～Fまでのアルファベットは小文字で書いても同じです。書式「%x」を「%X」とXを大文字にすると、大文字で出力されます。

第 3 日

複数回勝負のジャンケンゲームを作ろう

ゲームプログラムの手はじめとして、第2日でジャンケンゲームを作りました。
コンピュータと対戦し、1回勝負のゲームです。
しかし、1回勝負ではおもしろくないので、今度は5回勝負のジャンケンゲームに改造してみましょう。
5回繰り返してジャンケンを行い、その結果から最終的に「○勝○敗○引き分け」と勝敗を判定して表示します。
さらにもうひとつ、野球拳ゲームも作ってみましょう。

今日作るプログラムについて

5回勝負のジャンケンゲームプログラム

第2日に作ったジャンケンゲームをもとに5回連続してジャンケンを行うプログラムを作ります。5回繰り返してジャンケンを行い、その結果から、最終的に「○勝○敗○引き分け」と勝敗を判定して表示します。

ゲームスタート

↓

プレイヤー（自分）が「グー」「チョキ」「パー」のどれを出すか決めて入力

↓

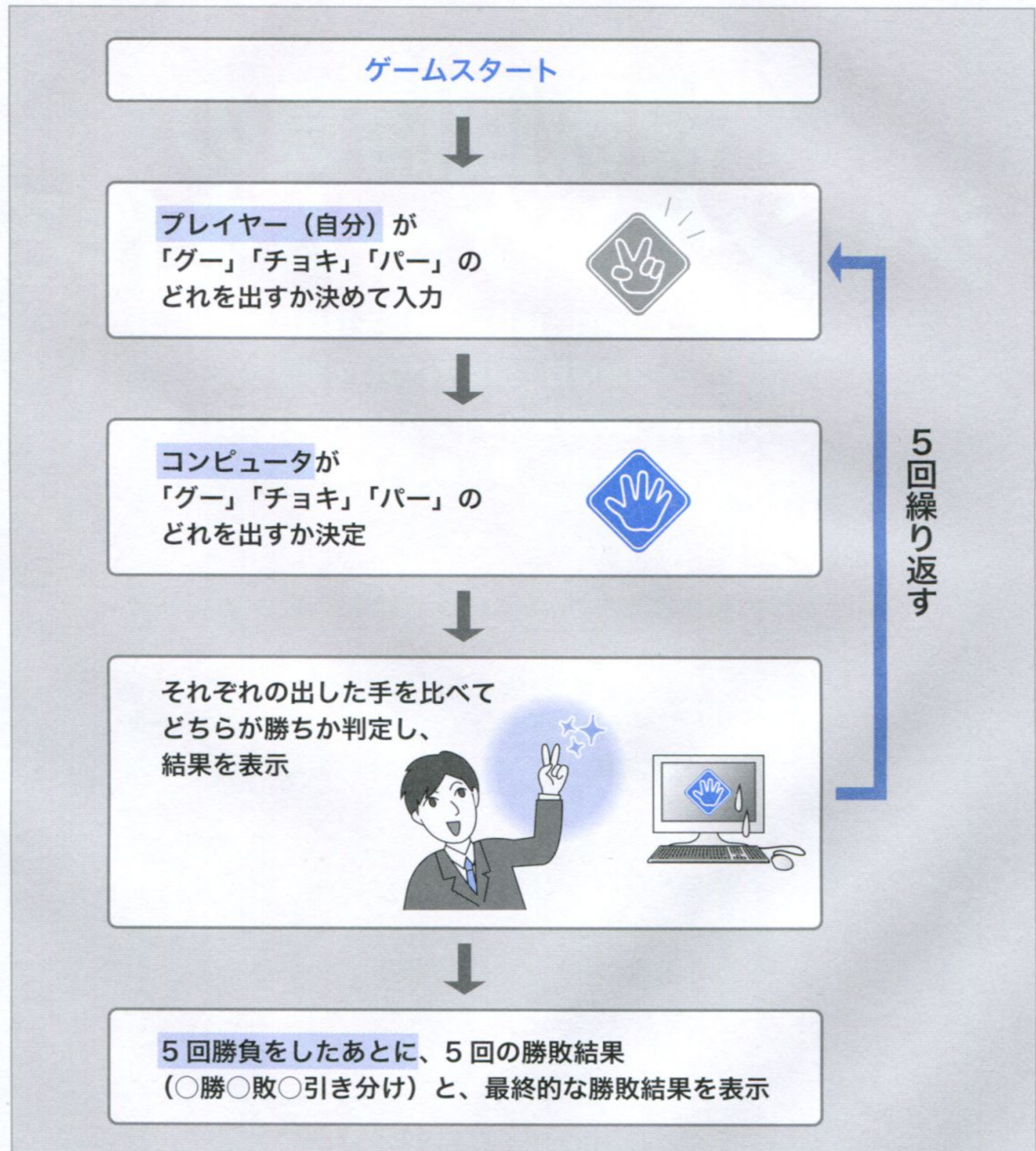
コンピュータが「グー」「チョキ」「パー」のどれを出すか決定

↓

それぞれの出した手を比べてどちらが勝ちか判定し、結果を表示

（5回繰り返す）

↓

5回勝負をしたあとに、5回の勝敗結果（○勝○敗○引き分け）と、最終的な勝敗結果を表示

５回勝負のジャンケンゲームの実際の動作

1 ジャンケンゲームプログラムを実行すると、プレイヤーの出す手を入力する状態になる

```
C:¥source>janken2.exe
【5 回勝負ジャンケンゲーム】
ジャンケン・・・( グー :1　チョキ :2　パー :3 を入力 ) >
```

2 プレイヤーがチョキを出す（数値の２を入力して、[Enter] キーを押す）

```
C:¥source>janken2.exe
【5 回勝負ジャンケンゲーム】
ジャンケン・・・( グー :1　チョキ :2　パー :3 を入力 ) > 2 ←「2」を入力する
```

3 コンピュータが手を出し、勝敗が決定して*1、次の勝負に移る

```
C:¥source>janken2.exe
【5 回勝負ジャンケンゲーム】
ジャンケン・・・( グー :1　チョキ :2　パー :3 を入力 ) > 2
コンピュータはチョキ！　あいこ
ジャンケン・・・( グー :1　チョキ :2　パー :3 を入力 ) >
```

ヒント

*1：1、2、3以外の数値を入力すると、その回はプレイヤーの負けになる。

4 上記①～③を５回繰り返す

5 ５回勝負がおわったところで、５回の通算成績が出る

```
（略）
ジャンケン・・・( グー :1　チョキ :2　パー :3 を入力 ) > 3
コンピュータはパー！　あいこ

2 勝 1 敗 2 引き分け　プレイヤーの勝利！
```

野球拳ゲームプログラム

繰り返しジャンケンを行い、先に5回負けた方が最終的に負けとなるゲームです。どちらかが5回負けるまで、ゲームは続きます。

ゲームスタート

↓

プレイヤー（自分）が「グー」「チョキ」「パー」のどれを出すか決めて入力

↓

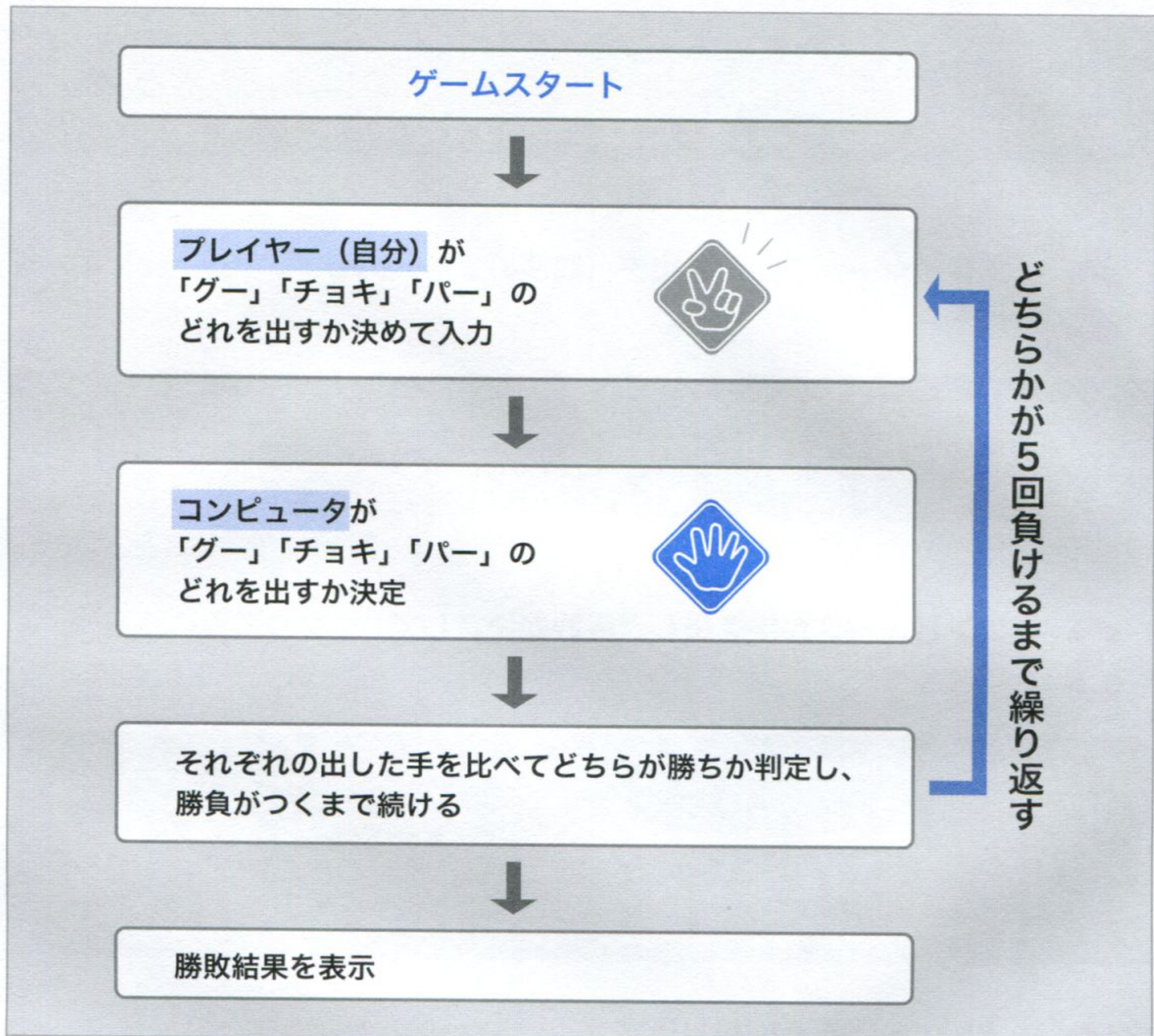
コンピュータが「グー」「チョキ」「パー」のどれを出すか決定

↓

それぞれの出した手を比べてどちらが勝ちか判定し、勝負がつくまで続ける

↓

勝敗結果を表示

どちらかが5回負けるまで繰り返す

野球拳ゲームの実際の動作

1 野球拳ゲームプログラムを実行すると、プレイヤーの出す手を入力する状態になる

```
C:¥source>yakyuuken.exe
【野球拳ゲーム】

やぁ～きゅうぅ～すーるならー
こういうぐあいにしやしゃんせ～
アウト　セーフ
よよいのよい！
（グー:1 チョキ:2 パー:3 を入力）>
```

2 プレイヤーがグーを出す（数値の１を入力して、[Enter] キーを押す）

```
（略）
( グー :1  チョキ :2  パー :3 を入力 ) > 1
```
←「1」を入力する

3 コンピュータが手を出し、勝敗が決定して*2、次の勝負に移る。あいこの場合は再勝負なので、もう一度手を出す（あいこでなくなるまで続ける）

```
（略）
( グー :1  チョキ :2  パー :3 を入力 ) > 1
コンピュータはパー！　コンピュータの勝ち
```

```
（略）
( グー :1  チョキ :2  パー :3 を入力 ) > 2
コンピュータはチョキ！　よよいのよい！
( グー :1  チョキ :2  パー :3 を入力 ) >
```

4 上記①～③を、プレイヤーかコンピュータのどちらかが５回負けるまで繰り返す

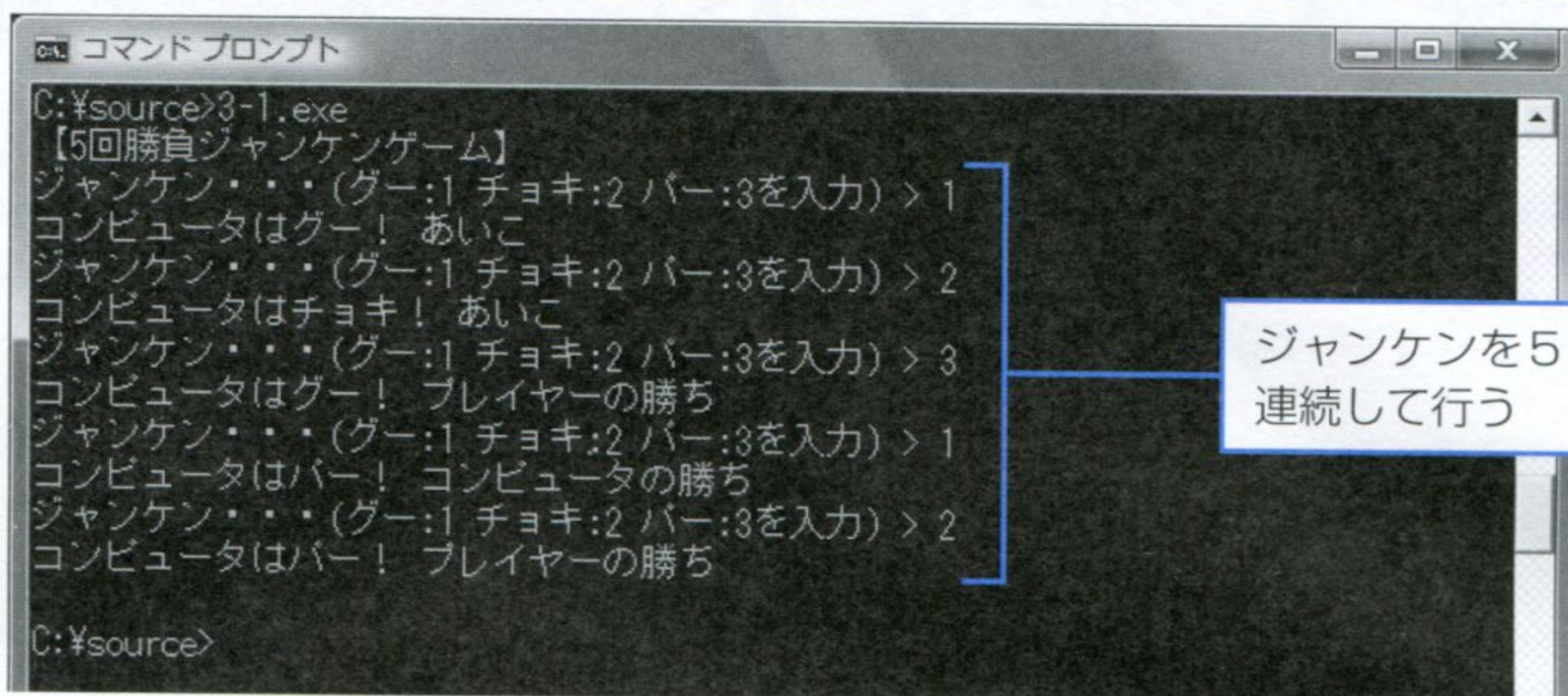
5 勝敗結果が表示される

```
（略）
よよいのよい！
( グー :1  チョキ :2  パー :3 を入力 ) > 1
コンピュータはチョキ！　プレイヤーの勝ち

勝負あり！
あなたの勝ち！
```

ヒント

*2：1、2、3以外の数値を入力すると、その回はプレイヤーの負けになる。

繰り返し処理を理解しよう

【複数回勝負のジャンケンゲームを作ろう①】

1時限目では繰り返し処理について学習し、ジャンケンを5回繰り返して行うプログラムを作ります。

今回作成する例題

```
C:¥source>3-1.exe
【5回勝負ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはグー! あいこ
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはチョキ! あいこ
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはグー! プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー! コンピュータの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはパー! プレイヤーの勝ち

C:¥source>
```

ジャンケンを5回連続して行う

サンプルファイルはこちら 10days_c ▶ day03-01 ▶ 3-1.c

●このレッスンのねらい

第2日に作ったジャンケンゲームのプログラムを応用します。
繰り返し回数が5回ときまっているので、ジャンケンゲームのプログラム部分を5回続けて書いてもよいのですが、それではプログラムとしてあまり美しくありません。
C言語では、同じ処理内容を繰り返し行うための文法が何種類か用意されているので、それを使いましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int player = 0, computer;
    int limit = 5; //繰り返し回数
    int i;

    printf("【%d回勝負ジャンケンゲーム】¥n",limit);
    srand(time(NULL));

    for(i = 0; i < limit; i++) {
        printf("ジャンケン･･･(グー:1 チョキ:2 パー:3を入力) > ");
        scanf("%d", &player);

        computer = rand()%3 + 1;
        printf("コンピュータは");
        if(computer == 1) { printf("グー"); }
        else if(computer == 2) { printf("チョキ"); }
        else { printf("パー"); }
        printf("！ ");

        if(computer == player) {
            printf("あいこ¥n");
        } else if(player == 1) {
            if(computer == 2) { printf("プレイヤーの勝ち¥n"); }
            else { printf("コンピュータの勝ち¥n"); }
        } else if(player == 2) {
            if(computer == 3) { printf("プレイヤーの勝ち¥n"); }
            else { printf("コンピュータの勝ち¥n"); }
        } else if(player == 3) {
            if(computer == 1) { printf("プレイヤーの勝ち¥n"); }
            else { printf("コンピュータの勝ち¥n"); }
        } else {
            printf("1,2,3のどれかを入力してね！¥n");
        }
    }
```

```
    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「3-1.c」という名前で[*1]、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、3-1.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 3-1 3-1.c
```

4 プログラムを実行する。ジャンケンを5回行うことができれば成功！

```
C:¥source>3-1.exe
```

▼

```
【5 回勝負ジャンケンゲーム】
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) > 1
コンピュータはチョキ！ プレイヤーの勝ち
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) > 2
コンピュータはパー！ プレイヤーの勝ち
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) > 3
コンピュータはチョキ！ コンピュータの勝ち
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) > 1
コンピュータはグー！ あいこ
ジャンケン・・・( グー :1 チョキ :2 パー :3 を入力 ) > 3
コンピュータはグー！ プレイヤーの勝ち
```

解説

1 for文を使った繰り返し処理

今回は5回勝負のジャンケンゲームを作ります。つまり、コンピュータとのジャンケンを5回繰り返して行います。

ジャンケンはまずプレイヤーの手を入力し、それからコンピュータの手をきめて、勝敗を判定しています。第2日に作ったプログラム*2のうち、

```
printf("ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > ");
      (略)
if(computer == player){
      (略)
} else {
    printf("1,2,3のどれかを入力してね！¥n");
}
```

この間のソースが、ジャンケン部分です。

5回勝負なので、この部分を5回繰り返して書く……のは非常に無駄です。繰り返し処理の文法を利用しましょう。

繰り返し処理の文法は、for文、while文、do～while文の、3種類があります。for文は、「5回繰り返す」など、繰り返しの回数がきまっている場合によく使います。対するwhile文は、繰り返し回数がきまっていない場合によく使います。

(1) for文の基本

for文の書き方は次のとおりです。

【for文】

```
for (初期化; 条件式; ループ処理) {
    処理;
}
```

条件式にあてはまっている間は、forブロックの中の処理を繰り返します。繰り返すためのしくみが、初期化とループ処理です。初期化は繰り返しに入る前に行い、ループ処理はブロックの中の処理がおわったら必ず実行する処理です。

実際にfor文を使ってみましょう。for文を使って、繰り返し処理を5回行います。繰り返し行う処理の内容は、出力処理だけです。

ヒント

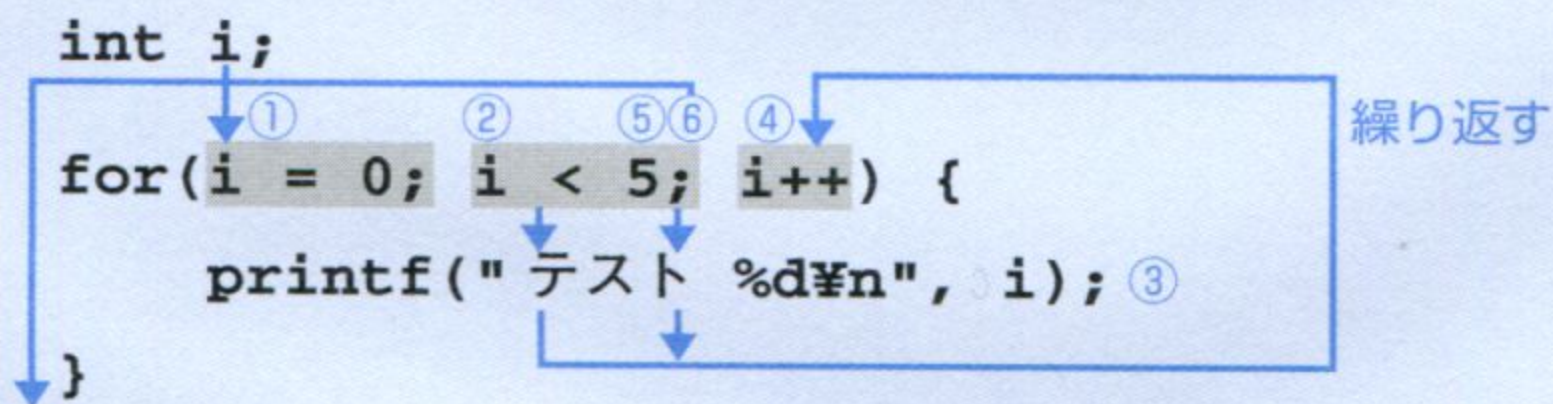
*2：第2日4時限目の練習問題の解答をふまえたプログラムとします。

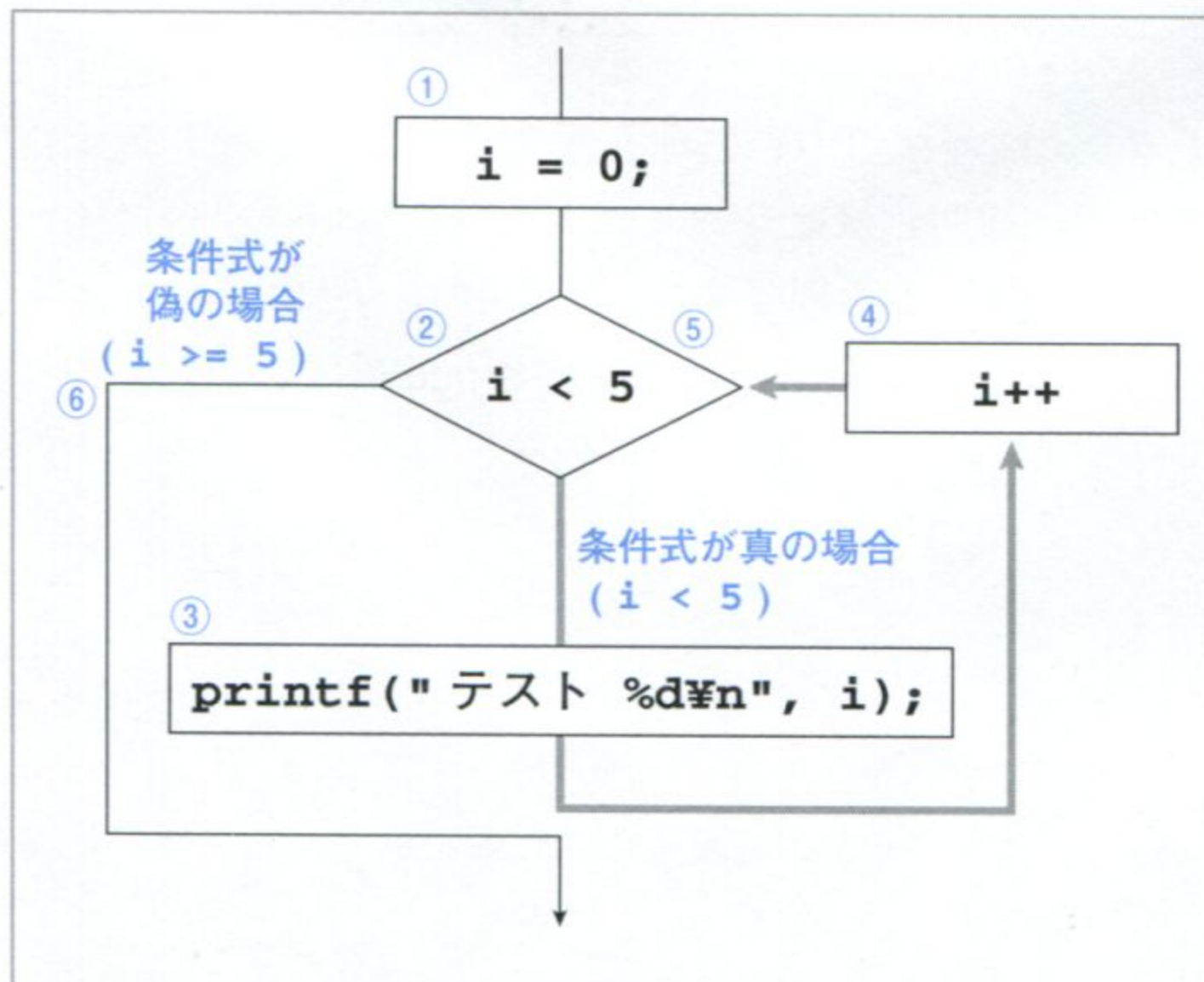
【3-1_sample1.c】

```
#include <stdio.h>

int main() {
    int i;

    for(i = 0; i < 5; i++) {
        printf("テスト %d¥n", i);
    }
    return 0;
}
```

```
テスト 0
テスト 1
テスト 2
テスト 3
テスト 4
```

この場合のforブロックの中の処理を図解すると、次のようになります。

```
int i;
    ①        ②       ⑤⑥  ④
for(i = 0; i < 5; i++) {
    printf("テスト %d¥n", i); ③
}
```

繰り返す

●3-1_sample1.cでのfor文の処理

for文は、「数値の変数が○から△になるまで繰り返す」という処理に使うのが基本です。この例の場合、変数iが0から5になるまで処理を繰り返す、という意味になります。少々面倒ですが、for文のすべての動作を追って見ていきましょう。

最初に、「int i;」と変数iを数値として宣言し、その次から、for文がはじまっています。

●3-1_sample1.cのfor文での実際の処理

・初期化（i = 0）を実行する。変数iに0を設定する。……①
・条件式(i < 5)を判定する。この時点でiは0なので、条件式にあてはまっている。よって、繰り返しを行う。……②
・forブロックの中身printf関数を実行する。iはこの時点で0。……③
出力→ テスト0
・forブロック終了。
・ループ処理（i++）[*3、*4]を行う。i++は、「iに1をプラスして、その値を再びiに代入する」という書き方。よってこのとき、iは1になる。……④
・条件式（i < 5）を判定。この時点でiは1なので、条件式にあてはまる。よって、続いて繰り返しを行う。……⑤
・forブロックのprintf関数を実行。
出力→ テスト1
・forブロック終了。ループ処理「i++」で、iは2になる。
・条件式判定。繰り返し続行。
・forブロックのprintf関数を実行。
出力→ テスト2
・forブロック終了。ループ処理「i++」で、iは3になる。
・条件式判定。繰り返し続行。
・forブロックのprintf関数を実行。
出力→ テスト3
・forブロック終了。ループ処理「i++」で、iは4になる。
・条件式判定。繰り返し続行。
・forブロックのprintf関数を実行。
出力→ テスト4
・forブロック終了。ループ処理「i++」で、iは5になる。
・条件式判定。iは5なので、条件式（i < 5）にはあてはまらない。よって、この時点で繰り返しは終了。……⑥

これだけの処理が、for文を使うとたった3行で書くことができます。

for文は便利な構文ですが、条件文やループ処理の書き方を間違うと繰り返し処理が終了せず、無限ループという状態に陥ってしまうので、注意しましょう。

(2) もしも無限ループになったら!?

こんなプログラムを書いてみます。

ヒント

*3：「i++」を、「iをインクリメントする」といいます。逆に、1をマイナスしてその値を再びiに代入する場合は「i--」と書きます。これを、「iをデクリメントする」といいます。
インクリメントやデクリメントする変数は、必ず最初に初期化を行いましょう。

ヒント

*4：繰り返しをカウントする変数には、iやjなど1文字の変数をよく使います。もちろん、countやtotalといった名前を付けてもかまいません。

```
int i;

for(i = 0; i < 5; i--) {
    printf("テスト %d¥n", i);
}
```

ループ処理が「i--」(デクリメント)なので、変数iは初期値0から次々に-1され、5以上になることは永遠にありません。すると、当然ながら繰り返し処理は永遠におわらずに、無限ループとなります。一度、試しにやってみましょう。実行してみると、すぐに無限ループに入り、「テスト XXXX」が出力され続けます。

とめるためには、コマンドプロンプト画面をアクティブにした状態で、[Ctrl]+[C]([Ctrl]キーを押しながら[C]キーを押す)を押すと、処理を中断できます。

(3) 変則的for文

for文の構文は基本的に、

```
for (初期化; 条件式; ループ処理) {
```

と書きますが、初期化、条件式、ループ処理は、それぞれ必ずしもすべて書く必要はありません。例えば、

```
int i = 0;
for(; i < 5; i--) {
```

と、初期化をfor文の前に行っているのであれば、for文の中で初期化は必要ありません。

また、次のようにループ処理をブロック中に含む方法もあります。

```
for(i = 0; i < 5;) {
    printf("テスト %d¥n", i++);
}
```

こういった書き方をすると、まず、現在のiの値がprintf関数で出力され、そのあとでiはインクリメントされます[*5]。

初期化、条件式、ループ処理のそれぞれが必要ない場合は、単純にその部分を書きません。3つを区別するセミコロン「;」は、すべて忘れずに書きます。

初期化、条件式、ループ処理を3つとも書かない場合は、

```
for(;;) {
```

となります。なお、これは無限ループになります。

ヒント

*5:逆に、printf("テスト %d¥n", ++i); と書くと、先にiがインクリメントされ、その値がprintf関数で出力されます。注意して使いましょう。

(4) 0からはじめるプログラムの世界

通常、私たちが何か数を数える場合、1、2、3、4……と、1から数えはじめます。「5回繰り返す」場合も、普通だったら1、2、3、4、5と数えます。ですが、先ほどのプログラムでは、0、1、2、3、4と数えていました。どちらも、回数でいえば同じ5回です。

先ほどのプログラムを1、2、3、4、5と数えるには、初期化と条件式の値を、それぞれ1増やします。次のように書くと出力表示は異なりますが、繰り返し回数は5回で同じです。

【初期化と条件式の値をそれぞれ１増やす】

```
for(i = 1; i < 6; i++) {
    printf("テスト %d¥n", i);
}
```

▼

```
テスト 1
テスト 2
テスト 3
テスト 4
テスト 5
```

こっちの方がわかりやすくていいじゃないか！ という人もいると思いますが、特に理由がなければ、コンピュータプログラムでは、数値を数えるときは0からはじめましょう。

2 繰り返しを強制的に終了する

繰り返し処理は、繰り返し条件に満たなくなったら繰り返しを終了します。しかし、繰り返し条件に関係なく途中で処理を終了する方法があります。それには、breakとcontinueを使います。

(1) break

第2日3時限目のswitch文のところでも出てきました。繰り返しブロック中に「break;」文が出てくると、そこで繰り返しを中断し、ブロックを抜けます。

次の例では、発生させた乱数が5の倍数だった場合、breakにより、繰り返しを5回待たずに、繰り返しを抜けます。

【3-1_sample2.c】

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i;
    int r;
```

```
    srand(time(NULL));
    for(i = 0; i < 5; i++) {
        r = rand();
        printf("%d 回目 ", i+1);
        if(r%5 == 0) { printf("5の倍数が出ました %d", r); break; }
        printf("%d¥n", r);
    }
    return 0;
}
```

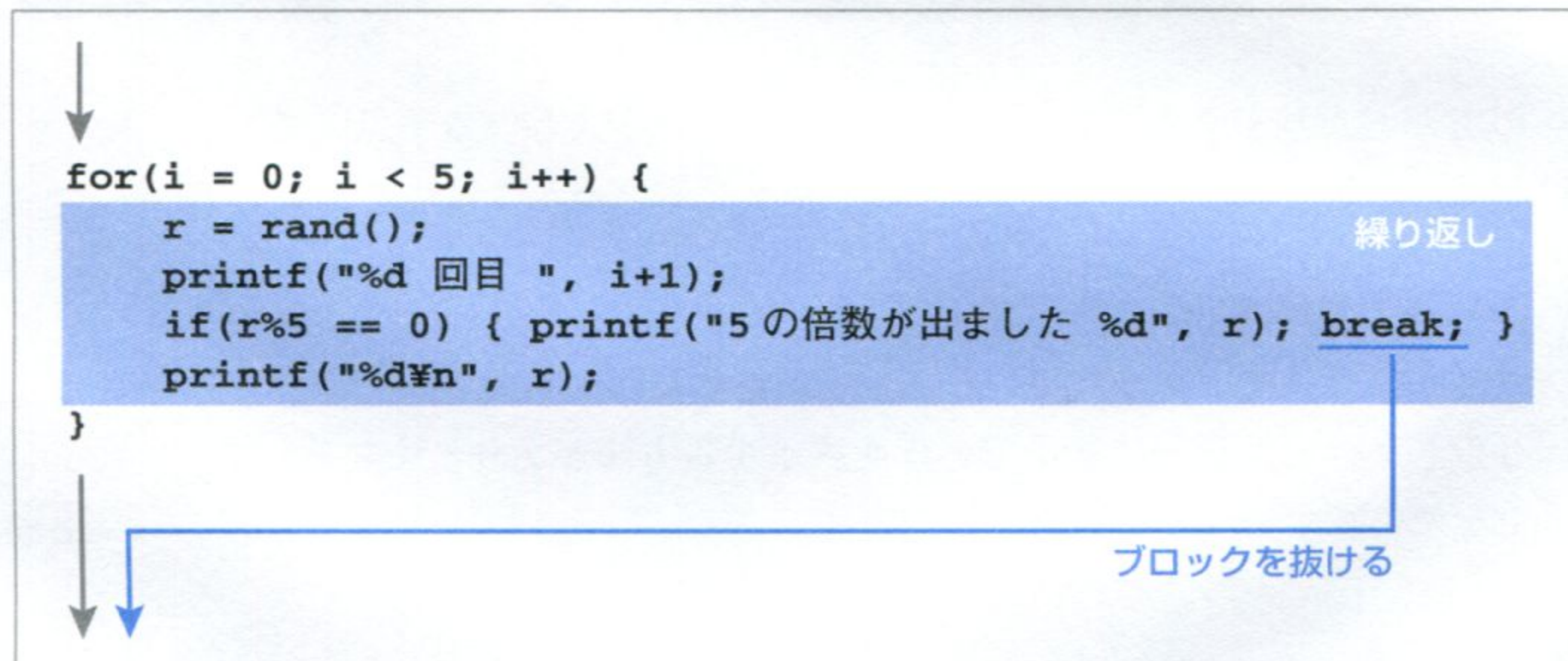
```
1 回目 41
2 回目 18467
3 回目 6334
4 回目 5の倍数が出ました 26500*6
```

ヒント
*6：実行結果は環境により異なります。

●breakで処理を抜ける

```
for(i = 0; i < 5; i++) {
    r = rand();
    printf("%d 回目 ", i+1);
    if(r%5 == 0) { printf("5の倍数が出ました %d", r); break; }
    printf("%d¥n", r);
}
```

繰り返し

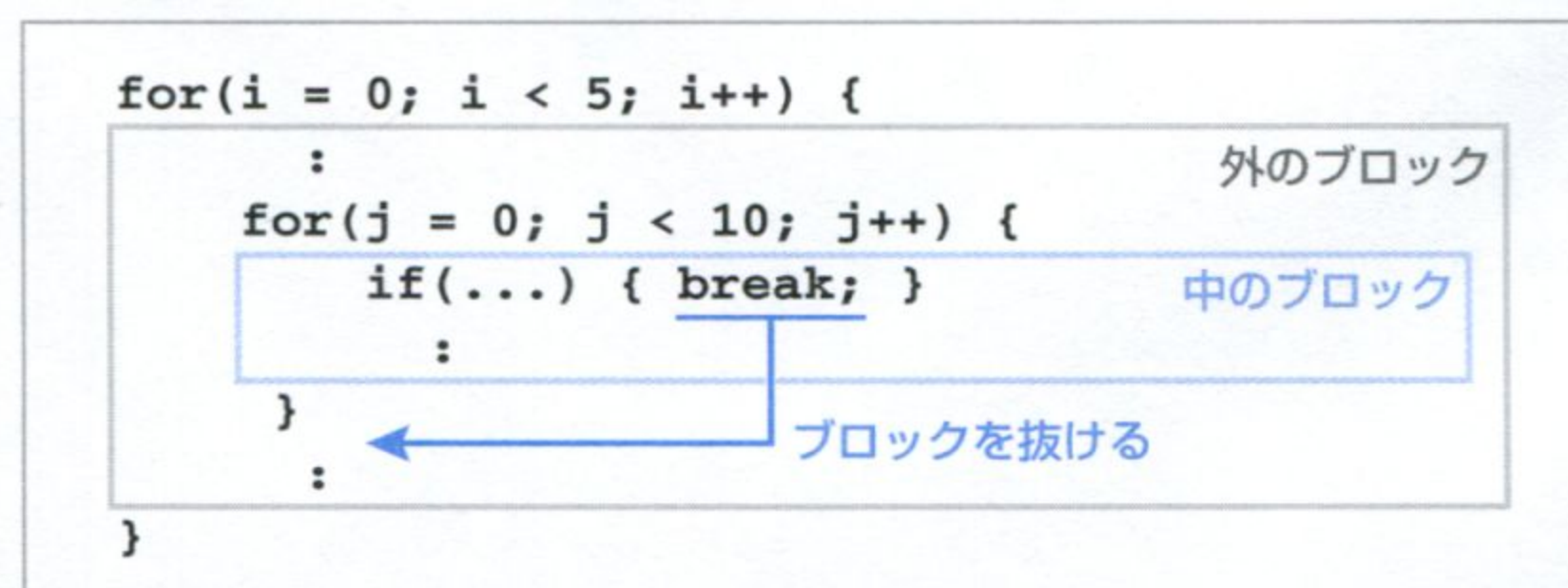
ブロックを抜ける

繰り返し処理中にさらに繰り返しブロックがある場合は、内側のブロックのみ、処理を抜けます。

●内側のブロックのみ処理を抜ける

```
for(i = 0; i < 5; i++) {
    :
    for(j = 0; j < 10; j++) {
        if(...) { break; }
        :
    }
    :
}
```

外のブロック

中のブロック

ブロックを抜ける

(2) continue

繰り返しブロック中に「continue;」文が出てくると、そこから先の処理を中断して繰り返しの先頭に戻ります。

次の例では発生させた乱数が5の倍数だった場合、先頭に戻ってループ処理を行います。

```
for(i = 0; i < 5; i++) {
    r = rand();
    printf("%d 回目 ", i+1);
    if(r%5 == 0) { printf("5の倍数が出ました %d¥n" ,r); continue; }
    printf("%d¥n", r);
}
```

```
1 回目 41
2 回目 18467
3 回目 6334
4 回目 5の倍数が出ました 26500*7
5 回目 19169
```

ヒント

*7：実行結果は環境により異なります。

●continueで繰り返しの先頭に戻る

```
for(i = 0; i < 5; i++) {
    r = rand();
    printf("%d 回目 ", i+1);
    if(r%5 == 0) { printf("5の倍数が出ました %d¥n" ,r); continue; }
    printf("%d¥n", r);
}
```

繰り返し

先頭に戻る

breakとcontinueは、for文、while文、do～while文のどの繰り返し処理中でも使うことができます。while文とdo～while文は4時限目で説明します。

3 ジャンケンの繰り返し

第2日に作ったジャンケンのプログラムを改造して、ジャンケンを5回繰り返して行うプログラム作成します。繰り返しの回数は5回ときまっているので、for文を使います。

```
int limit = 5; //繰り返し回数
int i;

printf("【%d回勝負ジャンケンゲーム】¥n",limit);

for(i = 0; i < limit; i++) {
  …
}
```

プログラム中での繰り返しの回数はすべてこのlimit変数を参照します。こうすることにより、あとで繰り返し回数を変更した場合でも最初の値設定を変更するだけですみ、他のプログラム部分には影響がありません。

こういった数値はなるべく変数にして、最初の変数宣言時に初期化しておきましょう。

まとめ

繰り返し処理のひとつであるfor文について学習しました。

これから作るゲームプログラムのほとんどで、繰り返し処理を行います。終了条件の設定が難しい場合は、P129のように動作を紙に書き出してみましょう。

COLUMN

C言語のコンパイラ

本書で使用するCコンパイラはMinGW日本版ですが、他にもフリーで入手可能なCコンパイラがあります。そのうちいくつかを紹介します。なお、それぞれのコンパイラの入手先や入手方法は変更になる場合が多いため、ここでは詳しく説明しません。他のコンパイラを使用してみたい人は、コンパイラの名前をGoogleなどの検索エンジンを使って検索し、最新版を入手してください[*8]。
本書の学習内容は付属CD-ROM収録のMinGW日本版の使用を前提としています。他のコンパイラでは、コンパイル方法や制約が少々異なる場合があります。他コンパイラの使用は自己責任でお願いします。

●Borland C++Compiler 5.5
ダウンロードには登録フォームへの入力が必要。登録後に送られてくるパスワードを使ってインストール実行。コンパイラ本体はbcc32.exe。

●MinGW
本書で使用しているコンパイラMinGW日本版のもとになっているコンパイラ。MinGW日本版のベースとなっているMinGWは少々古いバージョンになる。本家MinGWでも日本語を扱うことは可能だが、そのためにはいくつか作業が必要。

●LSI C-86 試食版
インストール場所にあわせてbin¥_LCCファイルの中身の書き換えが必要。コンパイラ本体はlcc.exe。

●Digital Mars C/C++ compiler for Win32
コンパイラ本体はsc.exe。

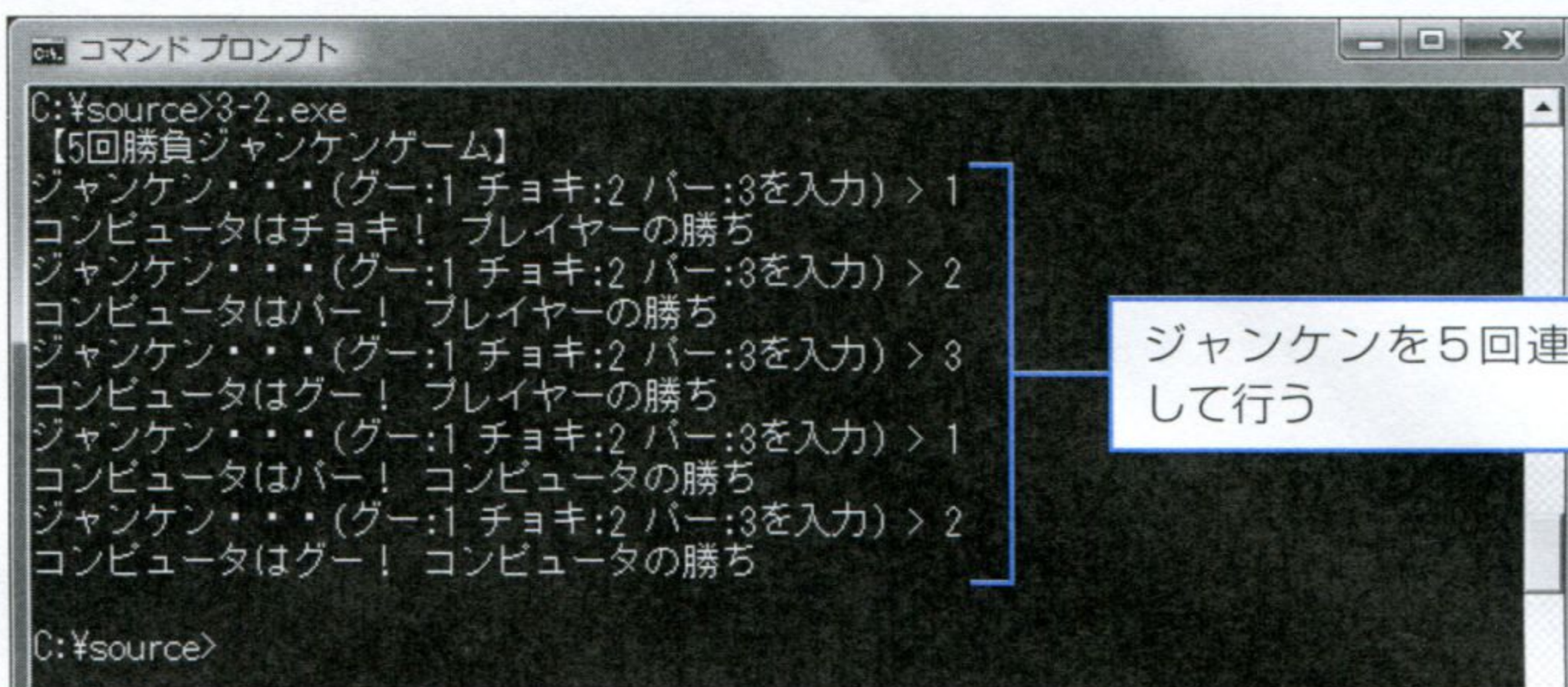
なお、ここまでに紹介したコンパイラは、すべてコマンドプロンプトを使って実行するものです。

●Cygwin - gcc
Cygwinとは、UNIXのさまざまなツールをWindowsで動作できるようにするもの。インストールしたCygwinを実行するとCygwin画面が立ち上がるので、そこでgccを実行する。

次の３つのコンパイラはIDE[*9]ソフトウェアです。ソフト内でコード編集・コンパイルが可能なので慣れれば便利ですが、使いこなすのは少々大変です。

●Microsoft Visual C++ 2008 Express Edition 日本語版
●LCC-Win32
●WideStudio/MWT

ヒント

*8：ここで紹介するコンパイラの入手および使用方法は2009年5月現在のものです。すべてインターネットのサイト上からダウンロードできます。

ヒント

*9：IDE（Integrated Development Environment）「統合開発環境」の略でエディタ、コンパイラ、デバッガなど、プログラミングに必要なツールがひとつのインターフェースで統合して扱えるような環境のこと。

第3日 2時限目
演算子について学ぼう
【複数回勝負のジャンケンゲームを作ろう②】

演算を行う識別子を演算子といいます。この時間は演算子について学習します。

今回作成する例題

```
C:¥source>3-2.exe
【5回勝負ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ! プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはパー! プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはグー! プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー! コンピュータの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはグー! コンピュータの勝ち

C:¥source>
```

ジャンケンを5回連続して行う

サンプルファイルはこちら 10days_c ▶ day03-02 ▶ 3-2.c

●このレッスンのねらい

演算子にはいろいろな種類があります。また、分岐や繰り返しの条件式で複数の条件を組みあわせて使うためには、論理演算子を利用します。
ここでは、今までに出てきた演算子について復習し、新たに論理演算子について学びましょう。演算子をうまく使うことができると、プログラムをスッキリと見やすくすることができます。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int player = 0, computer;
    int limit = 5; //繰り返し回数
    int i;

    printf("【%d回勝負ジャンケンゲーム】¥n", limit);
    srand(time(NULL));

    for(i = 0; i < limit; i++) {
        printf("ジャンケン…(グー:1 チョキ:2 パー:3を入力) > ");
        scanf("%d", &player);

        computer = rand()%3 + 1;
        printf("コンピュータは");
        if(computer == 1) { printf("グー"); }
        else if(computer == 2) { printf("チョキ"); }
        else { printf("パー"); }
        printf("！");

        if(computer == player){
            printf("あいこ¥n");
        } else if(player == 1 && computer == 2){
            printf("プレイヤーの勝ち¥n");
        } else if(player == 2 && computer == 3){
            printf("プレイヤーの勝ち¥n");
        } else if(player == 3 && computer == 1){
            printf("プレイヤーの勝ち¥n");
        } else {
            printf("コンピュータの勝ち¥n");
        }
    }
    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「3-2.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、3-2.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 3-2 3-2.c
```

4 プログラムを実行する。3-1.exeと同様に、5回ジャンケンを行うことができれば成功！

```
C:¥source>3-2.exe
```

▼

```
【5回勝負ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ！ プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはパー！ プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはチョキ！ コンピュータの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはグー！ あいこ
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはグー！ プレイヤーの勝ち
```

解説

1 演算子

演算子とは、何らかの動作を指定する記号のことです。「+」や「-」といった数値の計算を行うための算術演算子や、分岐処理の条件式で使う「==」や「>=」などの比較演算子があります。

今回のプログラムで使いたい論理演算子は、複数の条件がすべて真か、またはどれかひとつでも真となる条件があるかどうかを判定したいときに、条件式の中で使います。

2 算術演算子

数値の計算を行うための演算子が、算術演算子です。

●算術演算子

演算子	意味	使い方
＋	加算	a＋b
-	減算	a - b
*	乗算	a * b
/	除算	a / b（a ÷ bの商）
%	余り	a % b（a ÷ bの余り）
＝	代入	c＝a
++	1プラス	a++ a＝a＋1と同等（インクリメントと呼ぶ）
--	1マイナス	a-- a＝a - 1と同等（デクリメントと呼ぶ）

(1) /と%

C言語での割り算は「商」と「余り」に分割されます。例えば「7 ÷ 2」は、

7 ÷ 2＝3（商）と余り 1

になるので、次の場合は「/」だと商が、「%」だと余りが変数に代入されます。

```
int s = 7 / 2; ←── 商３を算出
int a = 7 % 2; ←── 余り１を算出
```

(2) 整数と実数

C言語のプログラムで、「/」演算子を使って、

5 / 2

を実行すると、結果は2になります。では、余りを使わずに「5 ÷ 2 = 2.5」という結果を出したい場合は、どうするのでしょう？ この場合、

5.0 / 2.0

と、このように実数で書くと、結果が2.5になります。次のように書いても同様です。

```
5.0 / 2
5 / 2.0
```

つまり、割られる数か割る数のどちらかが実数表記になっていれば、実数値の結果を取得できます。整数同士で計算すると、結果を受け取る変数が実数型でも、その結果は整数のときと同等の値になってしまいます。実数での計算結果を整数型で受け取っても同じことがいえます[*2]。

ヒント

*2：せっかく実数で割った結果を整数の変数に代入してしまうと、結果はまるめられてしまいます。
int a = 5.0/2.0;
この場合、int型変数aの値は2になります。注意しましょう。

```
double d = 5/2;   ← dの値は2.000000
```

(3) インクリメントとデクリメント

「++」や「--」は、値の計算と代入を同時に行っています。

```
a++    a+1を行ってその値をaに代入
a--    a-1を行ってその値をaに代入
```

このとき、「++a」「--a」と演算子を先に書くこともできます。意味は同じですが、演算子を変数の前に書くかうしろに書くかで、値をインクリメント、またはデクリメントするタイミングが異なります。

【演算子をうしろに書く場合】

```
int a = 1;
printf("%d", a++);
```

上の例では、aの現在の値を出力してから、インクリメントを行います。よって、出力値は1です。

【演算子を前に書く場合】

```
int a = 1;
printf("%d", ++a);
```

こちらの例の場合は、aをインクリメントしてから、その値を出力しています。よって、出力値は2です。この演算子を他の文と同時に使うときは、演算子の位置に注意しましょう。

3 代入演算子

「+」「-」「*」「/」「%」の算術演算子に続いて「=」を書くと、算術結果をそのまま代入する、代入演算子になります。

●代入演算子

演算子	意味	使い方
+=	足して代入	a += 2（a = a + 2と同等）
-=	引いて代入	a -= 2（a = a - 2と同等）
*=	掛けて代入	a *= 2（a = a * 2と同等）
/=	商を代入	a /= 2（a = a / 2と同等）
%=	余りを代入	a %= 2（a = a % 2と同等）

4 比較演算子

比較演算子は、分岐処理や繰り返し処理の条件式として使います。
比較演算子の結果が正しければ「真」、結果が間違っていれば「偽」となります。

●比較演算子

演算子	意味	使い方
==	イコール	a == b（aとbは等しい）
!=	イコールではない	a != b（aとbは等しくない）
>	大きい	a > b（aはbより大きい）
<	小さい	a < b（aはbより小さい）
>=	以上	a >= b（aはb以上）
<=	以下	a <= b（aはb以下）

5 論理演算子

複数の条件文をつなげる役目をするのが、論理演算子です。

●論理演算子

演算子	意味	使い方
&&	論理積	条件a && 条件b(条件aと条件b両方が真なら真、それ以外は偽)
||	論理和	条件a || 条件b（条件aと条件bのどちらか一方が真なら真、それ以外は偽)
!	否定	!a（aが真なら偽に、偽なら真にする）

変数iが10より大きく18よりも小さい場合の条件式は、次のようになります。

```
(i > 10) && (i < 18)*3
```

この条件とは逆に、変数iが10以下または18以上の場合の条件式は、次のようになります。

```
(i <= 10) || (i >= 18)*4
```

また、先ほどの10より大きく18よりも小さい条件式は10<i<18と似せて

ヒント
*3：それぞれの条件式は、なるべく( )で括りましょう。

ヒント
*4：比較演算子の左辺と右辺を逆に書いても同じです。

```
(10 < i) && (i < 18)
```

という書き方をする場合もあります。どちらも意味は同じです。

論理演算子でつなぐ条件式は、いくつでも増やすことができます。

```
条件a || 条件b || 条件c
```

これは、条件a、b、cのどれかひとつが真なら、真になります。
逆に条件a、b、cすべてが真の場合のみ真になるには、「&&」を使います。

```
条件a && 条件b && 条件c
```

条件式を書く順番にも注意しましょう。論理和の場合、条件aが真なら、条件bは真偽どっちでも結果は真になるので、条件bの判定は行いません。

```
条件a || 条件b
```

論理積の場合も考えてみましょう。こちらは条件aが偽なら、条件bの結果がどうであれ偽になるので、条件bの判定は行いません。条件aが真の場合のみ、条件bの判定を行います。

```
条件a && 条件b
```

どうでもよいことのように感じるかもしれませんが、実は結構重要です。おぼえておきましょう。

6 ジャンケンの判定をする

論理演算子について理解できたところで、今まで作成したジャンケンプログラムの判定部分を見直してみます。あいこの場合の他は、次のような分岐を行っています。

```
playerが1の場合
        computerが2の場合
        computerが3の場合
        (1の場合はすでに「あいこ」の条件にあてはまる)
```

playerの値が2、3の場合も、それぞれ同じ条件分岐を行っています。

論理演算子を使うと、もう少しスッキリ書くことができるのではないでしょうか。あいこの条件は同じですが、playerが1かつcomputerが2の場合はプレイヤーの勝ち、です。このように、論理演算子を使ってプレイヤーが勝ちの場合だけを抜き出し、それ以外の場合をコンピュータの勝ちとします。

また、プレイヤーが1、2、3以外の手を出した場合は、無条件にコンピュータの勝ちとするように変更しましょう。

これでだいぶ見やすくなります。

【論理演算子を使って勝敗を判定する】

```
if(computer == player){
    printf("あいこ¥n");
} else if(player == 1 && computer == 2){
    printf("プレイヤーの勝ち¥n");
} else if(player == 2 && computer == 3){
    printf("プレイヤーの勝ち¥n");
} else if(player == 3 && computer == 1){
    printf("プレイヤーの勝ち¥n");
} else {
    printf("コンピュータの勝ち¥n");
}
```

まとめ

演算子にはさまざまな種類があります。

ここでは、使い方に注意すべき演算子と、今日作るプログラムで使用する論理演算子についてのみ、詳細に説明しました。

他の演算子については、それぞれの演算子を使った例文を自分で作り、実際の動作を確認してみてください。

練習問題

Q

この時限で作成した論理演算子を使ってプログラムを作り直した部分を、さらに「||」を使って書き直しなさい。

解答は巻末に

第3日 3時限目

【複数回勝負のジャンケンゲームを作ろう③】5回勝負のジャンケンゲームを実行しよう

ここでは、1時限目と2時限目で作成した繰り返しの勝敗結果から、最終的な勝敗を判定する部分を作成して、5回勝負のジャンケンゲームを完成させます。

今回作成する例題

```
C:¥source>janken2.exe
【5回勝負ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ！ プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはパー！ プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはパー！ あいこ
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー！ コンピュータの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはチョキ！ あいこ

2勝1敗2引き分け　プレイヤーの勝利！

C:¥source>
```

5回ジャンケンを行い

最後に勝敗を表示する

サンプルファイルはこちら　10days_c ▶ day03-03 ▶ janken2.c

●このレッスンのねらい

1回勝負のときと同様に、勝敗結果はプレイヤーが勝つか、コンピュータが勝つか、または引き分けの3種類です。

プレイヤーの勝った数、負けた数、あいこの数を記憶しておき、最後にトータルの勝敗結果を表示します。

また、プレイヤーが不正な手を出したときの処理を追加して、ゲームを完成させます。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int player, computer;
    int p_win = 0; //プレイヤーの勝ち数をカウントする変数
    int p_lose = 0; //プレイヤーの負け数をカウントする変数
    int p_draw = 0; //引き分けの数をカウントする変数
    int limit = 5; //繰り返し回数
    int i;

    printf("【%d回勝負ジャンケンゲーム】¥n", limit);
    srand(time(NULL));

    for(i = 0; i < limit; i++) {
        player = 0;
        printf("ジャンケン…(グー:1 チョキ:2 パー:3を入力) > ");
        scanf("%d", &player);
        while (getchar() != '¥n') {  }

        computer = rand()%3 + 1;
        printf("コンピュータは");
        if(computer == 1) { printf("グー"); }
        else if(computer == 2) { printf("チョキ"); }
        else { printf("パー"); }
        printf("！ ");

        if(computer == player){
            printf("あいこ¥n"); p_draw++;
        } else if(player == 1 && computer == 2){
            printf("プレイヤーの勝ち¥n"); p_win++;
        } else if(player == 2 && computer == 3){
            printf("プレイヤーの勝ち¥n"); p_win++;
        } else if(player == 3 && computer == 1){
            printf("プレイヤーの勝ち¥n"); p_win++;
        } else {
```

```
            printf("コンピュータの勝ち¥n");
            p_lose++;
        }
    }

    printf("¥n%d勝%d敗%d引き分け ", p_win, p_lose, p_draw);

    if(p_win == p_lose) { printf("引き分け！¥n"); }
    else if(p_win > p_lose) { printf("プレイヤーの勝利！¥n");
}
    else { printf("コンピュータの勝利！¥n"); }
    return 0;
}
```

ヒント
*1：拡張子に注意して保存しましょう。

2 入力できたら、「janken2.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、janken2.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o janken2 janken2.c
```

4 プログラムを実行する。5回勝負の判定結果まで表示されれば成功！

```
C:¥source>janken2.exe
```

▼

```
【5回勝負ジャンケンゲーム】
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ！ プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはグー！ コンピュータの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはグー！ プレイヤーの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー！ コンピュータの勝ち
ジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 2
```

```
コンピュータはグー！　コンピュータの勝ち

2勝3敗0引き分け　コンピュータの勝利！
```

解説

1 勝敗の判定

繰り返しのジャンケンゲームを完成させましょう。

1時限目と2時限目で作ったジャンケンを5回繰り返すプログラムに、最終的な勝敗を決定する機能を加えます。『○勝△敗□引き分け　プレイヤー（またはコンピュータ）の勝利』と判定します。

5回勝負の間、プレイヤーの勝敗数をカウントしておいて、最後にその数値から結果を判定します。

●判定基準

プレイヤーの勝ち数と負け数が同じ	→	引き分け
プレイヤーの勝ち数が負け数よりも多い	→	プレイヤーの勝ち
プレイヤーの負け数が勝ち数よりも多い	→	コンピュータの勝ち

プレイヤーの勝ち数をカウントする変数をp_win、プレイヤーの負け数をカウントする変数をp_lose、引き分けの数をカウントする変数をp_drawとします。それぞれint型で、最初に0で初期化しておきます。

```
int p_win = 0; //プレイヤーの勝ち数をカウントする変数
int p_lose = 0; //プレイヤーの負け数をカウントする変数
int p_draw = 0; //引き分けの数をカウントする変数
```

ジャンケンを行い、プレイヤーが勝ったときにはp_winをインクリメントします。負け、あいこのときも同様に、それぞれの変数をインクリメントします。

5回の繰り返し終了後、p_win、p_lose、p_drawの数をそれぞれ判定して、最終的な勝敗を決定します。

```
printf("¥n%d勝%d敗%d引き分け  ", p_win, p_lose, p_draw);

if(p_win == p_lose) { printf("引き分け！¥n"); }
else if(p_win > p_lose) { printf("プレイヤーの勝利！¥n"); }
else { printf("コンピュータの勝利！¥n"); }
```

2 数値以外の入力について

プレイヤーの手を入力するとき、数値以外のものを入力すると、どうなるでしょうか？ 簡単なプログラムを作って、最初に「a」という文字を入力してみます。

【3-3_sample1.c】

```
#include <stdio.h>

int main() {
    int player = 0;
    int i;

    for(i = 0; i < 5; i++) {
        scanf("%d", &player);
        printf("%d 回目の値：%d¥n", i+1, player);
    }
    return 0;
}
```

```
a  ←入力
1 回目の値：0
2 回目の値：0
3 回目の値：0
4 回目の値：0
5 回目の値：0
```

ジャンケンを行う際、5回とも数値を入力すれば何の問題もありません。

しかし、文字を入力したら、5回繰り返さなければならないのに、1文字入力しただけで、あっという間にプログラムがおわってしまいました。出力された「X 回目の値：0」はplayerの初期値です。つまりscanf関数で、何も読み込まれていないことになります。

第2日2時限目でも述べましたが、scanf関数は、書式指定[*2]で期待していなかった入力データに出会うと、読み込みを中止してしまいます。

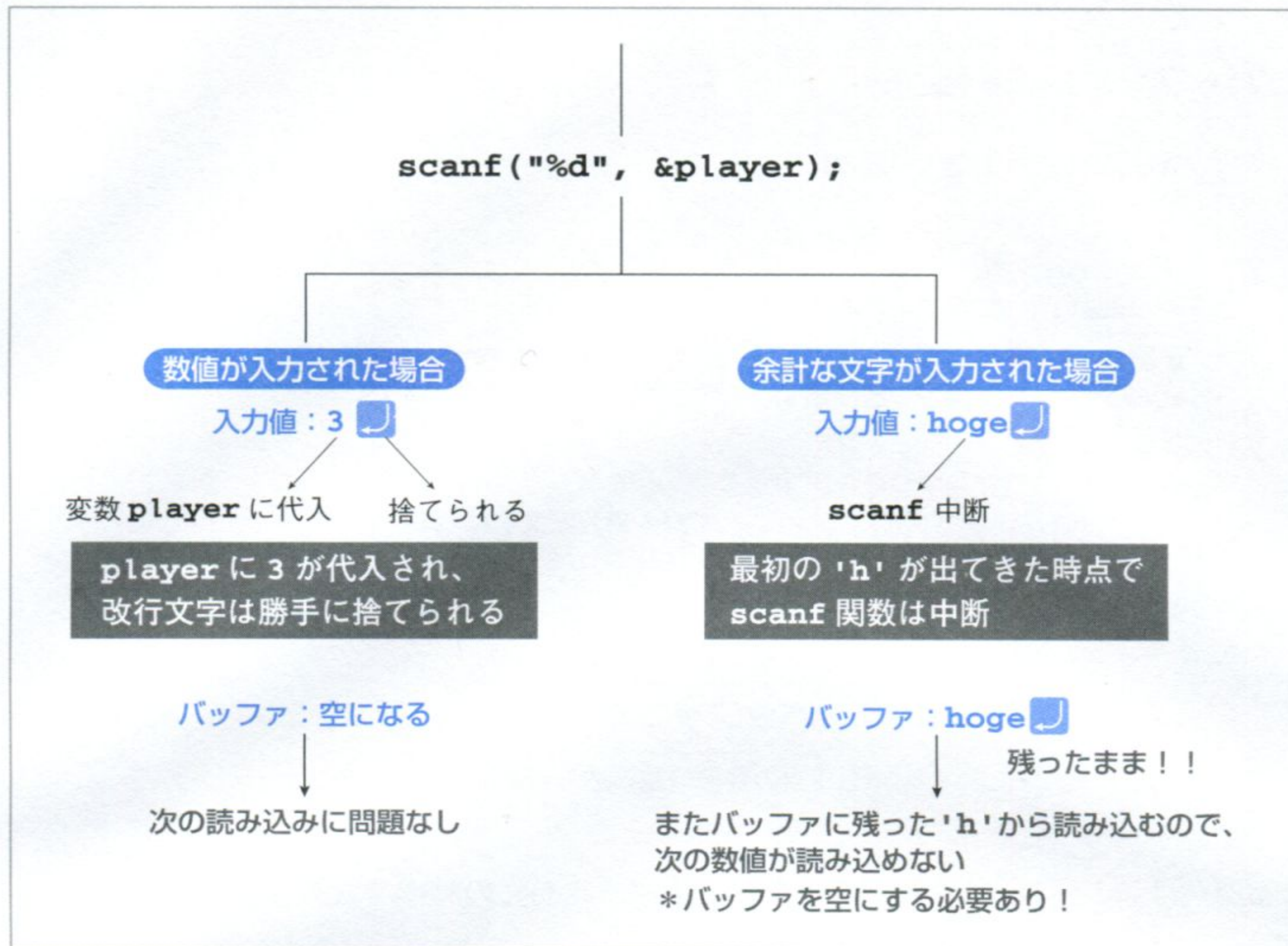
よって、先のプログラムでは、数値を入力してもらうはずだったのに、文字「a」が入ってきたのでscanf関数は終了し、次の繰り返しでも、前に読み込んだ文字が残っているために同じ動作になり、ずっと次の数値を読み込むことができません。

これを解消するために、間違ったデータが入力された場合、その入力値から改行までを捨てる処理を組み込まなくてはなりません。

これも第2日2時限目で述べたように、キーボードからの入力は、バッファと呼ばれる領域に格納されます。scanf関数やgetchar関数などの入力関数は、このバッファから入力値を取り出します。

ヒント

*2：scanf関数の%d書式では、次の数字が来るまで空白文字（改行を含む）を勝手に捨てるので、普通に数値のみを入力した場合は、次のscanf関数も正常に実行することができるのです。

●scanf関数がバッファから入力値を取り出すしくみ

入力データは何文字あるかわかりません。よって、このときはwhile文を使い、数値以外のデータが入力されたら、そのあとは1文字ずつ読み込みを行って、最後に改行を表す「¥n」が来るまで、処理を繰り返します。

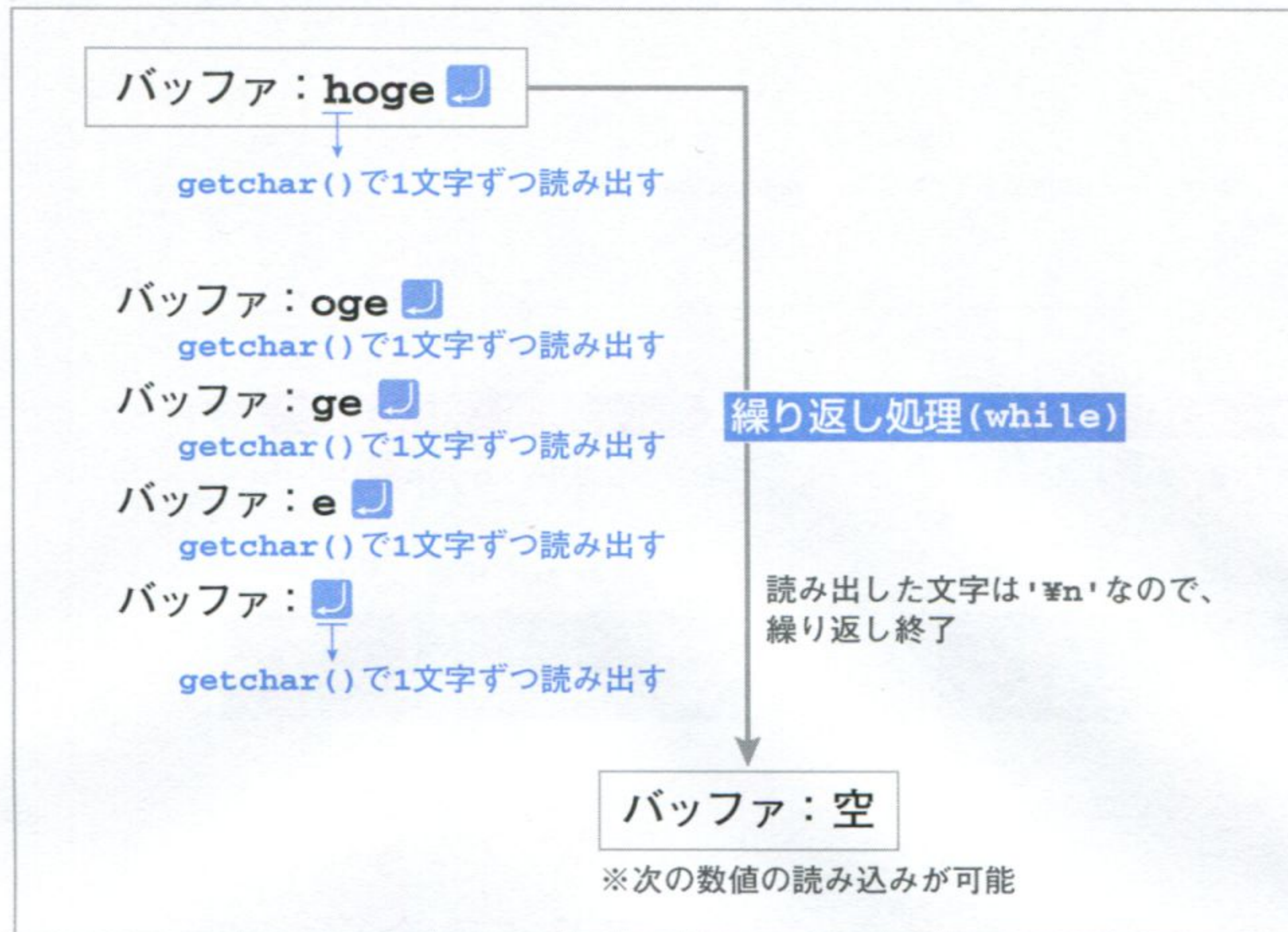
なお、while文については次の4時限目にじっくり説明します。ここでは繰り返しのジャンケンゲームを完成させるために、説明前ですが使用してしまいます。「while(条件) { 処理}」として、条件が真の間だけ処理ブロックを繰り返します。

では、scanf関数のあとに、次の1行を入れましょう。

```
while (getchar() != '¥n') {  }
```

getchar関数で繰り返し読み込みを行っているので、whileブロック中の処理は何も必要ありません。

最後の「¥n」を取り出してしまえばバッファは空になるので、次の数値を読み込むことができます。

●getchar関数がバッファから値を読み出すしくみ

scanf関数は読み込みに成功すると1を返します。while文の動作を含め、確認してみましょう。

【3-3_sample2.c】

```
#include <stdio.h>

int main() {
    int player = 0;
    int i;
    int r;

    for(i = 0; i < 5; i++) {
        r = scanf("%d", &player);
        while (getchar() != '¥n') {  }
        printf("%d 回目の値：", i+1);
        if(r) { printf("%d¥n", player); } // 入力が正常の場合
        else { printf("入力エラー¥n"); } // 入力が不正な場合
    }
    return 0;
}
```

```
2          ←――― 入力値
1 回目の値：2
3a         ←――― 入力値 *3
2 回目の値：3
p          ←――― 入力値
3 回目の値：入力エラー
hoge       ←――― 入力値
4 回目の値：入力エラー
4          ←――― 入力値
5 回目の値：4
```

ヒント

*3：「3a」と入力した場合、数値の3を読み込んだ次の文字「a」でscanf関数が中断になります。一応数値が読み込まれたことになるので、scanf関数は1を返します。

もうひとつ注意しなければならないのは、間違ったデータが入力された場合、格納するための変数playerには、何も入ってこないということです。

scanf関数が中断されたとき、変数playerには前の入力値（または初期値）が残っている状態なので、これを毎回入力前に初期化するようにしましょう。

繰り返しブロックの最初に、次の行を入れます。

```
player = 0;
```

こうすれば、読み込む前に変数playerが、毎回0に設定されます。数値が入力されれば、変数playerはその数値にかわりますが、数値以外が入力された場合は0のままです。

0の場合、無条件にプレイヤーの負けです。もしもプレイヤーが本当に0を入力しても、それも入力ミスなので、プレイヤーの負けになります。

まとめ

5回繰り返すジャンケンゲームを作りました。

プレイヤーの入力値対策も追加したので、ゲームとして正確なものになったと思います。

scanf関数の直後のwhile文の使い方はこのあともずっと出てきますので、ここで意味をかんたんに理解しておいてください。

ここでは、繰り返す回数がきまっていないジャンケンゲームを作ります。

今回作成する例題

```
C:¥source>yakyuuken.exe
【野球拳ゲーム】

やぁ～きゅうぅ～すーるならー
こういうぐあいにしやしゃんせ～
アウト　セーフ
よよいのよい！
(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー！ コンピュータの勝ち

やぁ～きゅうぅ～すーるならー
こういうぐあいにしやしゃんせ～
アウト　セーフ
よよいのよい！
(グー:1 チョキ:2 パー:3を入力) > 2
コンピュータはグー！ コンピュータの勝ち

やぁ～きゅうぅ～すーるならー
こういうぐあいにしやしゃんせ～
アウト　セーフ
よよいのよい！
(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはグー！ プレイヤーの勝ち

やぁ～きゅうぅ～すーるならー
```

```
やぁ～きゅうぅ～すーるならー
こういうぐあいにしやしゃんせ～
アウト　セーフ
よよいのよい！
(グー:1 チョキ:2 パー:3を入力) > 3
コンピュータはチョキ！ コンピュータの勝ち

勝負あり！
あなたの負け！
C:¥source>
```

どちらかが5回負けるまでジャンケンを行い、最後に勝敗を表示する

サンプルファイルはこちら　10days_c ▶ day03-04 ▶ yakyuuken.c

●このレッスンのねらい

3時限目で作成した5回勝負のジャンケンゲームプログラムを応用し、本レッスンでは野球拳ゲームを作ります。野球拳は、繰り返し回数のきまっていないジャンケンゲームです。プレイヤーとコンピュータでジャンケンをして、どちらかが5回負けるまで勝負を続けます。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int player, computer;
    int p_count = 5;// プレイヤーの負け数残り
    int c_count = 5;// コンピュータの負け数残り

    printf("【野球拳ゲーム】¥n");
    srand(time(NULL));
    do {
        printf("¥nやぁ～きゅうぅ～すーるならー¥n");
        printf("こういうぐあいにしやしゃんせ～¥n");
        printf("アウト　セーフ¥n");
        do {
            printf("よよいのよい！¥n");
            printf("(グー:1 チョキ:2 パー:3を入力) > ");
            player = 0;
            scanf("%d", &player);
            while (getchar() != '¥n') { }
            computer = rand()%3 + 1;
            printf("コンピュータは");
            if(computer == 1) { printf("グー"); }
            else if(computer == 2) { printf("チョキ"); }
            else { printf("パー"); }
            printf("！ ");
        } while((player != 0) && (computer == player));

        if((player == 1 && computer == 2) ||
                (player == 2 && computer == 3) ||
                (player == 3 && computer == 1)) {
            printf("プレイヤーの勝ち¥n");
            c_count--;
        } else {
            printf("コンピュータの勝ち¥n");
            p_count--;
```

```
        }
    } while((p_count != 0) && (c_count != 0));

    printf("¥n 勝負あり！ ¥n");
    if(p_count == 0) {
        printf(" あなたの負け！ ");
    } else {
        printf(" あなたの勝ち！ ");
    }
    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「yakyuuken.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、yakyuuken.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o yakyuuken yakyuuken.c
```

4 プログラムを実行する。どちらかが5回負けるまで繰り返しジャンケンを行い、勝敗結果が表示されれば成功！

```
C:¥source>yakyuuken.exe
```

▼

```
【野球拳ゲーム】

やぁ～きゅうぅ～すーるならー
こういうぐあいにしやしゃんせ～
アウト　セーフ
よよいのよい！
( グー :1 チョキ :2 パー :3 を入力 ) > 1
コンピュータはパー！ コンピュータの勝ち
```

```
(略)
よよいのよい！
( グー :1  チョキ :2  パー :3 を入力 ) > 1
コンピュータはチョキ！　プレイヤーの勝ち

勝負あり！
あなたの勝ち！
```

解説

1 while文を使った繰り返し処理

繰り返し処理の、もうひとつの代表的なものは、while文です。

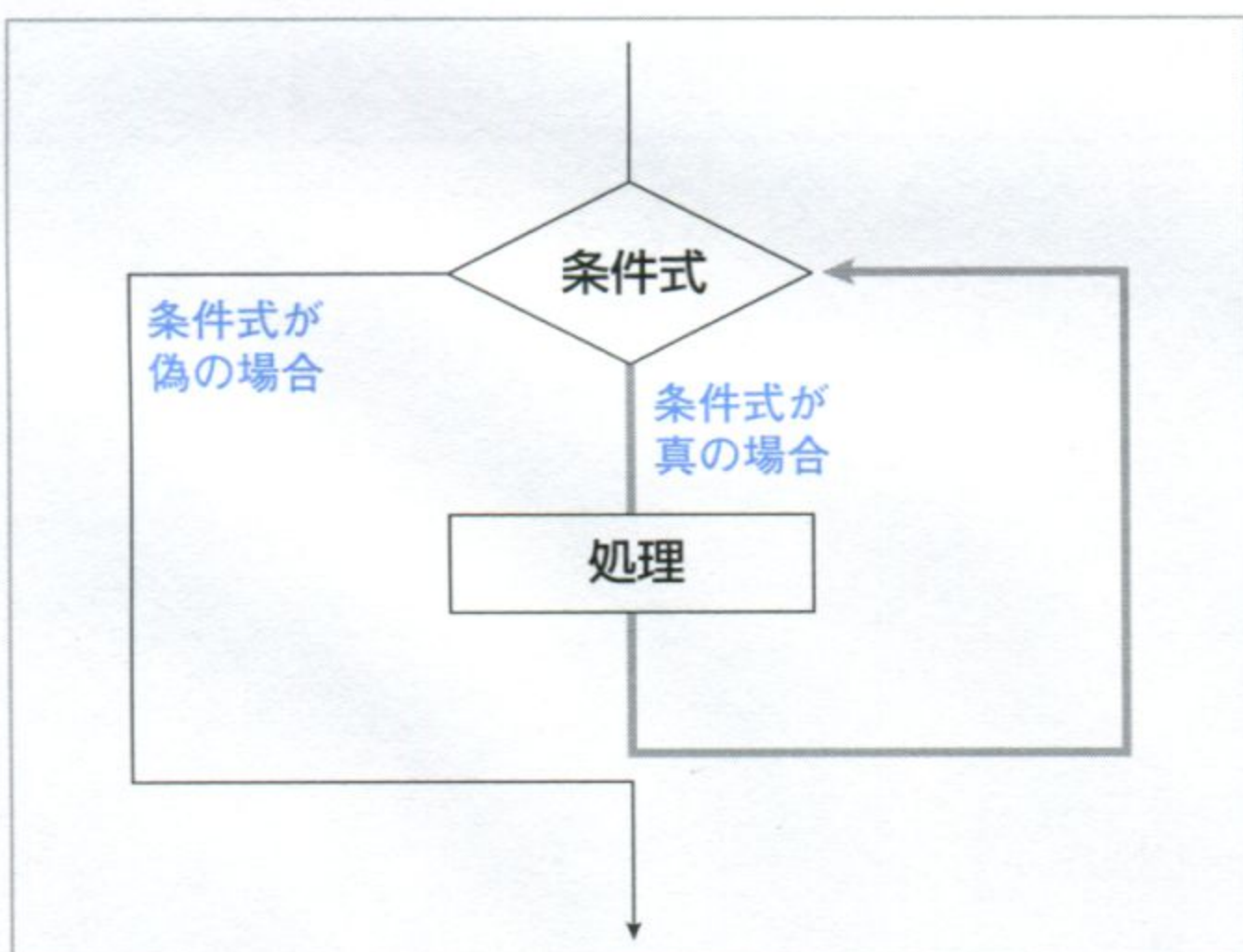
(1) while文の基本

while文の書き方は次のとおりです。

【while文】

```
while ( 条件式 ) {
    処理 ;
}
```

●while文の処理

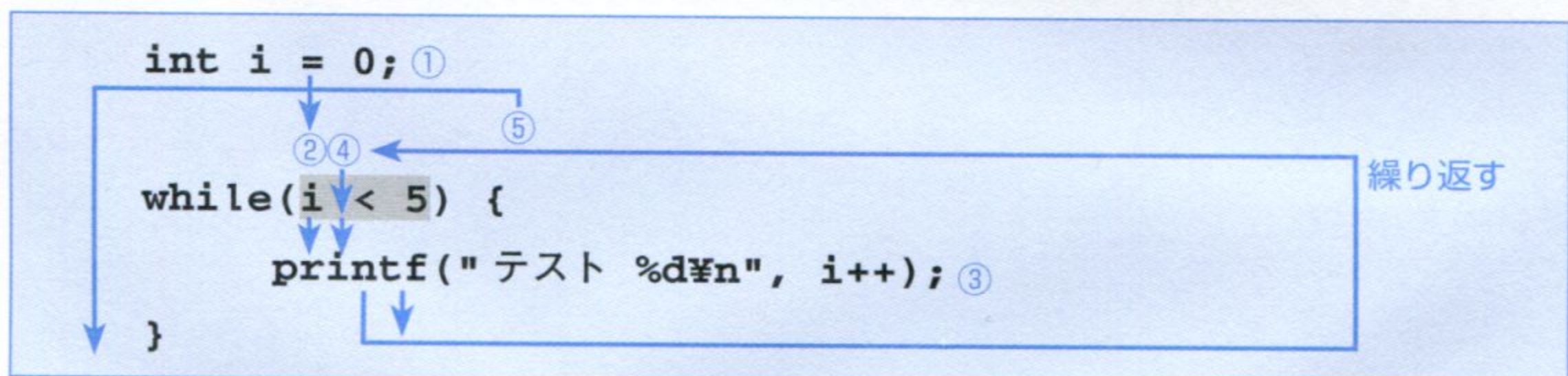
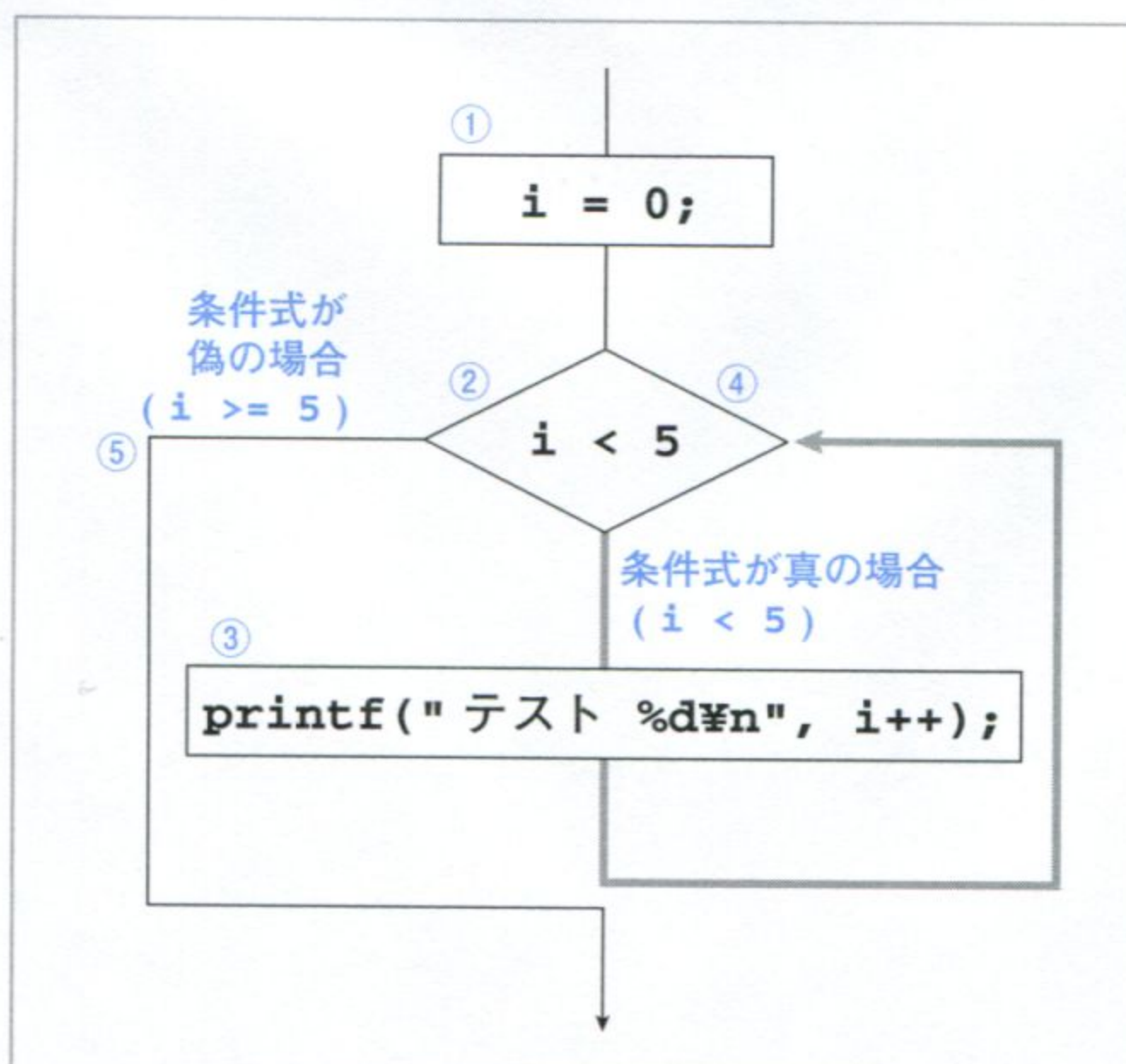
条件式
条件式が偽の場合
条件式が真の場合
処理

条件式にあてはまっている間は、whileブロックの中の処理を繰り返します。

for文を使って5回繰り返しを行うプログラムを、今度はwhile文を使って書いてみましょう。

【3-4_sample1.c】

```
#include <stdio.h>

int main() {
    int i = 0;

    while(i < 5) {
        printf("テスト %d¥n", i++);
    }
    return 0;
}
```

```
テスト 0
テスト 1
テスト 2
テスト 3
テスト 4
```

```
int i = 0; ①
            ⑤
  ②④
while(i < 5) {                          繰り返す
    printf("テスト %d¥n", i++); ③
}
```

●3-4_sample1.cでのwhile文の処理

for文と比較すると、初期化を先に行い、ループ制御をブロックの中に含めたものと同様です。

●3-4_sample1.cのwhile文での実際の処理

・最初に数値の変数iを0で初期化しておく。……①
・while文の条件式（i＜5）を判定する。この時点でiは0なので、条件式にあてはまる。
　よってwhileブロックの中身を実行する。……②
・whileブロックの中身printf関数を実行する。iはこの時点で0。……③
　出力→ テスト0
　出力後、iはインクリメントされて1になる。
・whileブロック終了。
・条件式判定。繰り返し続行。……④
・whileブロックのprintf関数を実行する。
　出力→ テスト1
　出力後、iは2になる。
・whileブロック終了。
・条件式判定。繰り返し続行。
・whileブロックのprintf関数を実行する。
　出力→ テスト2
　出力後、iは3になる。
・whileブロック終了。
・条件式判定。繰り返し続行。
・whileブロックのprintf関数を実行する。
　出力→ テスト3
　出力後、iは4になる。
・whileブロック終了。
・条件式判定。繰り返し続行。
・whileブロックのprintf関数を実行する。
　出力→ テスト4
　出力後、iは5になる。
・whileブロック終了。
・条件式判定。iは5なので条件式（i ＜ 5）にあてはまらない。よって、この時点で繰り返しは終了。……⑤

このプログラムの実行結果は、for文を使ったプログラムとまったく同じです。

for文と同様、while文でも無限ループにならないように注意しましょう。このサンプルでは、繰り返しは5回行うときまっていますが、最初に説明したように、while文は何回繰り返すかきまっていない場合に、よく使います。

(2) while文の書き方

while文では、for文と違って条件式を書かないとエラーになります。無限ループを作る場合、

```
while(1) {
```

と書きます。条件式が1で固定なので、永遠に繰り返し処理を行います。

(3) 入力バッファを空にする

3時限目の最後に、数値以外の値が入力された場合の処理として、scanf関数のあとに次のwhile文を入れました。getchar関数で読み込んだ文字が「¥n（改行）」でない間は繰り返し読み込みを行う、という処理です*2。

```
while (getchar() != '¥n') {  }
```

このように、処理ブロック部分に処理を書かないこともできます。

ヒント

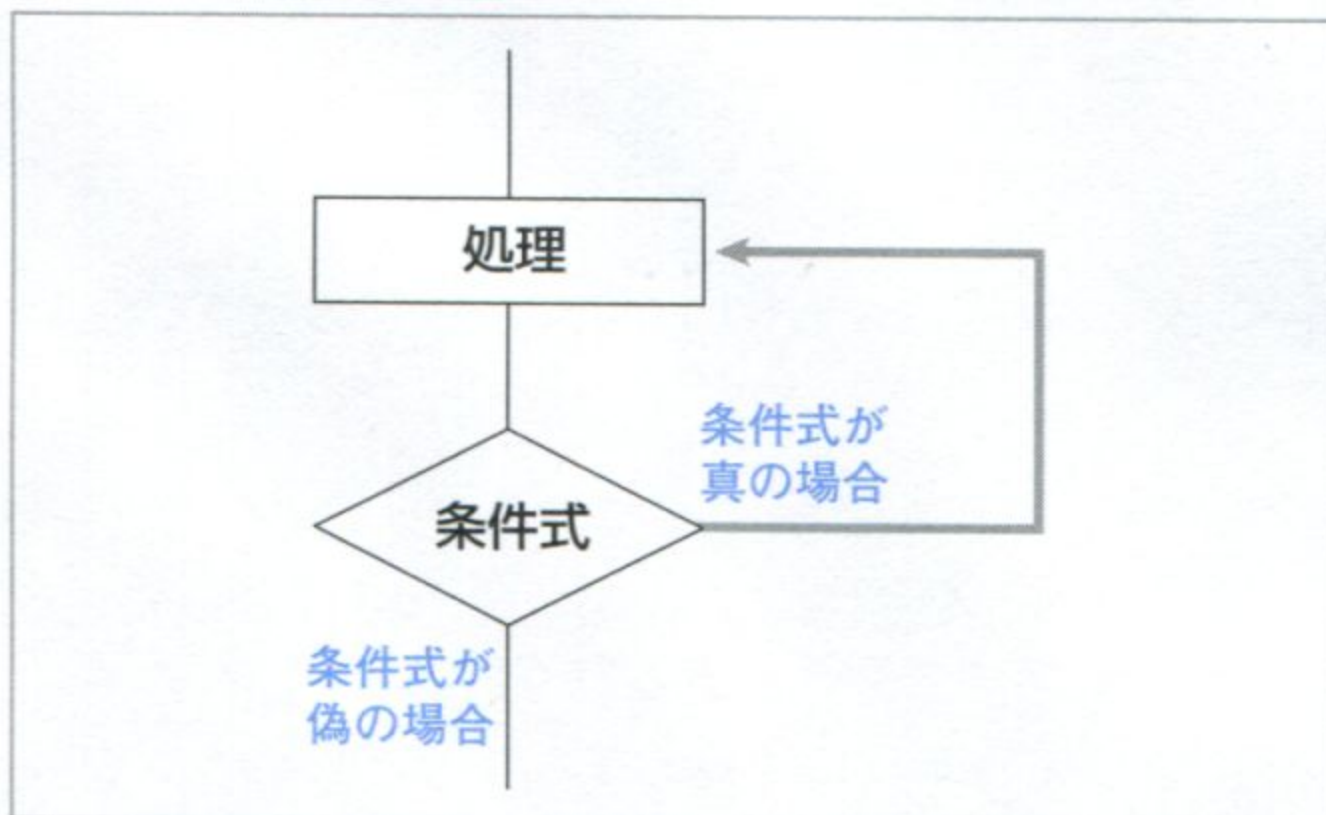
*2：入力された値はバッファと呼ばれる領域に一時的に格納されます。数値以外の値が入力されたらscanf関数は中断され、バッファに残ったデータによっては次の入力に影響が出るので、読み込むごとにこの処理を入れて、バッファを空にします。詳細については3時限目をご覧ください。

2 do～while文を使った繰り返し処理

do～while文の使い方は、基本的にwhile文と同じです。書き方は次のとおりです。

【do～while文】

```
do {
    処理 ;
} while ( 条件式 );
```

●do～while文の処理

条件式にあてはまっている間は、whileブロックの中の処理を繰り返します。while文と違って条件式が最後に来ているので、最低一度は、無条件でブロックの中の処理を行うことになります。

次のプログラムを試してみましょう。

【3-4_sample2.c】

```
#include <stdio.h>

int main() {
    int d;

    do {
        printf(" 数値を入力 > ");
        scanf("%d", &d);
    } while(d != 0);
    return 0;
}
```

3-4_sample2.cは、数値を続けて入力し、0が入力されたら繰り返しを終了するプログラムです。動作を確認しましょう（文字や文字列が入力された場合の処理は省略します）。

最初に、無条件でdo～whileブロックを実行します。

●3-4_sample2.cのdo～while文の実際の処理

・do～whileブロックのprintf関数とscanf関数を実行。入力した値を変数dに格納。
・do～whileブロック終了。
・条件式（d != 0）を判定。この時点でdが0でなければ、処理を繰り返す。
・再びdo～whileブロックのprintf関数とscanf関数を実行。
　この時点で、0が入力されたとする。
・do～whileブロック終了。
・条件式を判定。この時点でdが0なので、繰り返しを終了。

プログラムを実行してみましょう。

```
数値を入力 > 100
数値を入力 > 123
数値を入力 > 4
数値を入力 > 0
```

0が入力されなければ、いつまでも入力を行うことになります[*3]。

ヒント
*3：while と do ～ whileでも、繰り返しを終了するbreakと、繰り返しブロックの最初に戻るcontinueを使うことができます。

3 野球拳ゲームを作る

5回勝負のジャンケンゲームを応用して、野球拳ゲームを作ってみましょう。

どちらかが5回負けるまでジャンケンを繰り返します。繰り返しの回数はきまっていないので、while文かdo～while文を使います。

まず、プレイヤーとコンピュータの、それぞれの負けの残り数を記録する変数を作っておきます。

```
int p_count = 5;// プレイヤーの負け数残り
int c_count = 5;// コンピュータの負け数残り
```

初期値は5で、負けたほうの数が1ずつ減ります。

また、野球拳ではあいこの場合は、もう一度手を出し直します。そこで、あいこにならなくなるまで両方の手をきめ続ける繰り返し処理を、入れ子で作ります。

```
do {
    do {
        プレイヤーの手の入力;
        コンピュータの手を決定;
    } while((player != 0) && (computer == player));
              あいこ（プレイヤーの入力ミスではなく）なら入力を繰り返す

    if((player == 1 && computer == 2) ||
            (player == 2 && computer == 3) ||
            (player == 3 && computer == 1)) {
        printf("プレイヤーの勝ち¥n");
        c_count--;
    } else {
        printf("コンピュータの勝ち¥n");
        p_count--;
    }
} while((p_count > 0) && (c_count > 0));
```

繰り返しが終了したら、最後に勝敗を判定します。

その時点で、負け数の残りが0になっている方が負けになります。

まとめ

while文とdo～while文を使って、野球拳ゲームを作りました。

構文さえ理解していれば、それほど難しいプログラムではありません。歌の部分の出力などは、自分で自由にかえてみてください。

練習問題

Q

ここで作成したプログラムで「3a」と入力すると、「3」と入力された扱いになる。これを不正な入力とし、その回はプレイヤーの負けとなるようにプログラムを変更しなさい。

解答は巻末に

第4日

脳トレゲームを作ろう

第４日では脳トレゲームを２つ作ります。
ひとつは単純な計算問題です。計算結果を答えるまでの時間を計測します。
もうひとつは、０～９までの数字のうち、ひとつだけ抜けている数値を当てるゲームです。こちらも解答までの時間を計測し、その判定結果を出力します。
C言語を学習しつつ、さらに作成したプログラムで脳を鍛えましょう。

今日作るプログラムについて

計算問題脳トレゲーム

「3+9」や「11-6」などの計算問題をコンピュータが出題し、その解答を入力して答えます。出題から解答するまでの時間を計測して、その合計時間から「優秀！」「普通」「遅い」を判定して表示します。
問題数は全部で10問です。

計算問題脳トレゲームの実際の動作

1 計算問題脳トレゲームプログラムを実行すると、計算問題が出題される

```
C:¥source>noutore1.exe
 9 + 16 =
```

2 答えを入力して［Enter］キーを押す

```
C:¥source>noutore1.exe
 9 + 16 = 25
```

←答えを入力して［Enter］キー

3 次の問題が出題される

```
C:¥source>noutore1.exe
 9 + 16 = 25
 4 + 12 =
```

←新たな問題が出題された

4 出題と解答を繰り返し、足し算に5問、引き算に5問に答える

```
20 + 10 = 30
17 + 4 = 21
1 + 2 = 3
10 + 8 = 18
12 + 4 = 16
```

```
22 − 14 = 8
18 − 11 = 7
19 − 16 = 3
20 − 16 = 4
24 − 8 = 16
```

5 **10題目の解答を入力すると、正解数と平均解答時間、合計時間からなる判定が表示される。なお、不正解の問題は、ペナルティとして解答時間に3秒プラスして表示される**

```
20 + 10 = 30
17 +  4 = 21
 1 +  2 = 3
10 +  8 = 18
12 +  4 = 16
22 − 14 = 8
18 − 11 = 7
19 − 16 = 3
20 − 16 = 4
24 −  8 = 16
10 問中 10 問正解！
平均解答時間 1.922
判定は・・・優秀！
```

数値当て脳トレゲーム

0〜9までの数字をランダムに並べ、ひとつだけ抜けている数値を答えとして入力します。こちらも解答までの時間を計測し、その合計時間から「優秀」「普通」「遅い」を判定して表示します。

```
750314869 ←── 2が抜けている
607524391 ←── 8が抜けている
498317652 ←── 0が抜けている
075632948 ←── 1が抜けている
365412079 ←── 8が抜けている
```

数値当て脳トレゲームの実際の動作

1 数値当て脳トレゲームプログラムを実行すると、0～9までの数値のうち、ひとつだけ抜けている状態で、かつ並びはランダムに表示される

```
C:¥source>noutore2.exe
750314869 =>
```

2 抜けてる数値を入力して、[Enter] キーを押す

```
C:¥source>noutore2.exe
750314869 => 2
```

答えを入力して [Enter] キー

3 次の問題が出題される

```
C:¥source>noutore2.exe
750314869 => 2
607524391 =>
```

新たな問題が出題された

4 出題と解答を繰り返す

```
402731856 => 9
748536029 => 1
175962348 => 5
705691284 => 3
416572089 => 1
549183620 => 7
068129734 => 5
917360245 => 8
209163754 => 8
041372965 => 8
```

5 **10題目の解答を入力すると、正解数と平均解答時間、合計時間からの判定が表示される。なお不正解の問題は、ペナルティとして解答時間に5秒プラスして表示される**

```
402731856 => 9
748536029 => 1
175962348 => 5
705691284 => 3
416572089 => 1
549183620 => 7
068129734 => 5
917360245 => 8
209163754 => 8
041372965 => 8
10 問中 8 問正解！
平均解答時間 5.859
判定は・・・普通です！
```

第4日 1時限目

配列を理解しよう

【脳トレゲームを作ろう①】

データを保存する入れ物として、「配列」という形式について学習します。

今回作成する例題

```
C:¥source>4-1.exe
値を入力してください > 1
値を入力してください > 2
値を入力してください > 3
値を入力してください > 4
値を入力してください > 5
値を入力してください > 6
値を入力してください > 7
値を入力してください > 8
値を入力してください > 9
値を入力してください > 10
1 回目の入力値：1
2 回目の入力値：2
3 回目の入力値：3
4 回目の入力値：4
5 回目の入力値：5
6 回目の入力値：6
7 回目の入力値：7
8 回目の入力値：8
9 回目の入力値：9
10 回目の入力値：10

C:¥source>
```

10回入力を行い…

その値を表示する

サンプルファイルはこちら　10days_c ▶ day04-01 ▶ 4-1.c

●このレッスンのねらい

今までの学習で扱った変数は、それぞれが独立しています。
同じ種類の変数を複数扱う場合、それらの変数をひとつのまとまりとして持つ「配列」という概念を使うことができます。
10問分の計算の正解と、それに対する解答、解答までの時間をそれぞれデータとして保存しておき、最後にまとめて判定します。そのために配列を利用します。
この1時限目では、配列を理解するために基本的な学習をしましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>

int main() {
    int kotae[10];   // 解答を入れる配列
    int i;
    int r;

    for(i = 0; i < 10; i++) {
        r = 0;
        printf(" 値を入力してください > ");
        scanf("%d", &r);
        while (getchar() != '¥n') {  }
        kotae[i] = r;
    }
    for(i = 0; i < 10; i++) {
        printf("%d 回目の入力値：%d¥n", i+1, kotae[i]);
    }

    return 0;
}
```

2 入力できたら、「4-1.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント
*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、4-1.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 4-1 4-1.c
```

4 プログラムを実行する。10回繰り返して値を入力したあとに、その値がすべて表示されれば成功！

```
C:¥source>4-1.exe
```

```
値を入力してください > 1
値を入力してください > 2
値を入力してください > 3
値を入力してください > 4
値を入力してください > 5
値を入力してください > 6
値を入力してください > 7
値を入力してください > 8
値を入力してください > 9
値を入力してください > 10
1 回目の入力値：1
2 回目の入力値：2
3 回目の入力値：3
4 回目の入力値：4
5 回目の入力値：5
6 回目の入力値：6
7 回目の入力値：7
8 回目の入力値：8
9 回目の入力値：9
10 回目の入力値：10
```

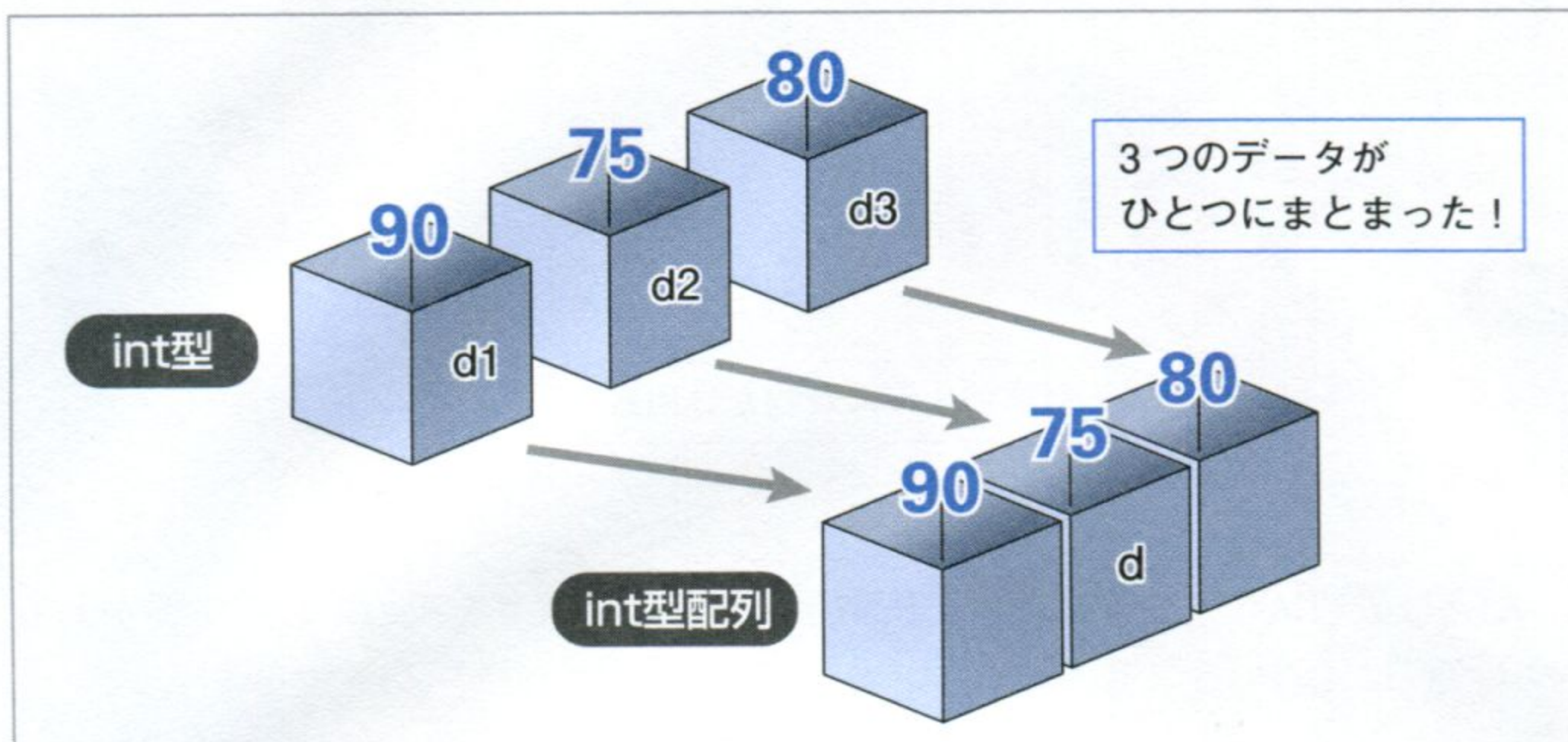
解説

1 配列を定義する

最初に、配列の基本を理解しましょう。
配列とは「データのあつまり」です。例えば、3つの数値変数を定義してみます。

```
int d1 = 90;
int d2 = 75;
int d3 = 80;
```

ここでは変数d1、d2、d3を、あるテストの成績としましょう。d1は1回目、d2は2回目、d3は3回目のテストの点数です。この3つは「テストの点数」というデータのあつまり、つまり配列として扱うことができます。

●配列はデータのあつまり

配列のデータは、次のように定義します。

【配列の定義】

```
int d[3] = { 90, 75, 80 };
```


変数をひとつひとつ定義するときと同様に、まずはデータの型を定義します*2。配列に入っている各データは、すべて定義した型のデータになります。この場合はすべてint型のデータです。

型のあとに続くのが、配列の変数名です。普通の変数名と同様、半角英数文字を使って自由に定義できます。

変数名に続いて[ ]がついていると、配列になります。[ ]の中に書いてある数値は、添え字といいます。これで配列の中に格納されているデータの数を定義します*3。

配列を定義するときに添え字を書かないと、初期値から勝手に定義されます。例えば次の場合は、勝手にd[3]となります。

【添え字を書かず、初期値で配列の大きさがきまる定義方法】

```
int d[] =  { 90, 75, 80 };
```

配列定義の＝の右側が配列の初期値です。値は{ }で括り、それぞれの値はカンマ「,」で区切ります。

初期値を最初に設定しない場合は、次のように配列の大きさだけを定義して、配列の入れ物だけを用意しておきます。

【初期値を書かず、配列の大きさだけをきめておく定義方法】

```
int d[3];
```

ヒント

*2：配列変数も、他の変数と同じくmain関数の最初にまとめて定義します。

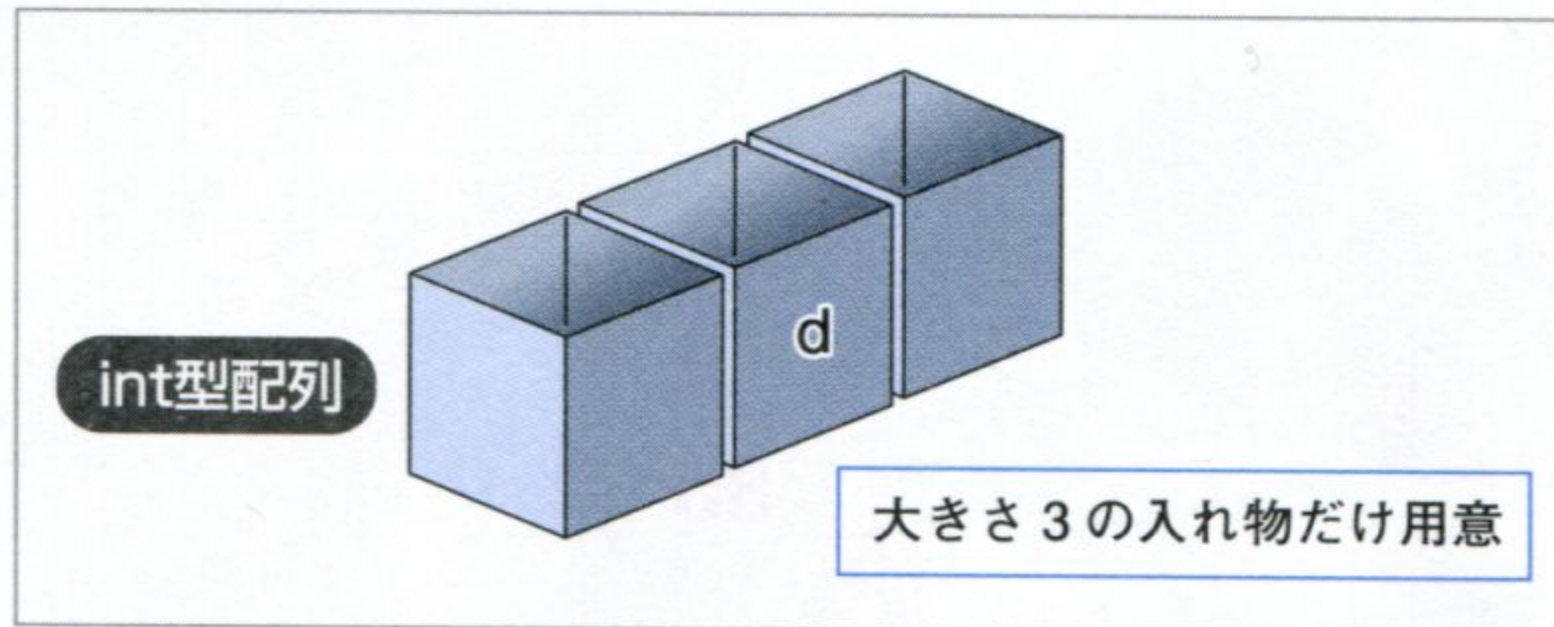
ヒント

*3：配列に格納されているデータの数を、「配列の大きさ」や「配列の長さ」ともいいます。また、配列の中のひとつひとつのデータを、「配列の要素」といいます。

●配列の大きさだけ定義

int型配列

d

大きさ３の入れ物だけ用意

しかし、配列は最初に「○個のデータを格納します」と大きさを決めておく必要があるため、次のようには定義できません。

```
int d[];
```

個数が不明な場合は、ある程度余裕を持った個数を適度に定義しておきましょう。いくつ用意しておくべきかわからない場合は、だいたいで予想した個数に近い128や256、1024など、2のべき乗の数値を用意しておくとよいでしょう。

2 配列のデータを参照・格納する

配列を定義する方法はわかりました。次に、配列の中の、ひとつひとつのデータの扱い方を学習します。

(1) 配列のデータを参照する

ここでは、先ほどの数値配列dを例にとります。

```
int d[3] = { 90, 75, 80 };
```

この配列の最初のデータ「90」を参照するには、

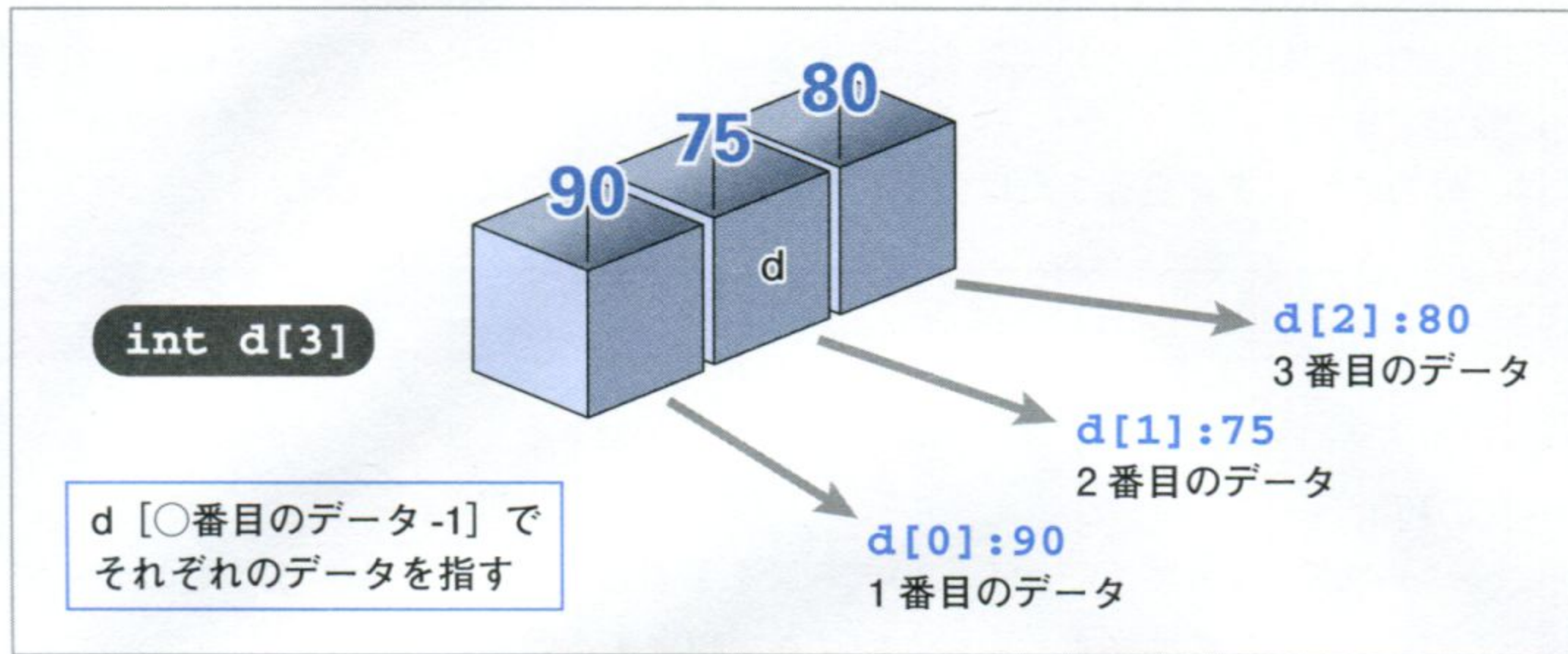
d[0]

添え字番号

配列変数名

と指定します。配列変数名に続いて[ ]を書き、その中に、前から何番目のデータなのか、番号を指定します。「最初のデータなのに添え字番号は0？」と疑問に思う方もいると思いますが、添え字は必ず0から数えます。おぼえておきましょう。

同様に、2番目のデータはd[1]、この例での最後のデータはd[2]です。

●配列の添え字と要素の関係

では、配列のそれぞれのデータを参照するプログラムを作ってみます。

【4-1_sample1.c】

```
#include <stdio.h>

int main() {
    int d[3] = { 90, 75, 80 };

    printf("d[0]:%d d[1]:%d d[2]:%d¥n", d[0], d[1], d[2]);
    return 0;
}
```

```
d[0]:90 d[1]:75 d[2]:80
```

それぞれのデータが参照されました。

(2) 配列にデータを格納する

次に、配列の値を変更したり、最初に初期化を行わなかったりした場合の値の定義の仕方を見てみましょう。普通の変数は、

```
int d;
d = 10;
```

とあとから値を定義できます。これを配列に応用してもよさそうなものですが、

```
int d[3];
d = { 90, 75, 80 };
```

とすると、これはNGです。コンパイルでエラーになります。

一度定義した配列には、まとめて値を設定することはできません。各データに、それぞれ個別に設定します。

例えば、最初のデータを定義または変更したい場合は、次のようにします。

```
d[0] = 95;
```

(3) for文の利用

配列はデータのあつまりです。すべてのデータを参照したり、あとからすべてのデータを挿入したりするのに、添え字番号をひとつひとつ指定するのは面倒です。

配列は単なるデータのあつまりではなく添え字番号0から順に並んでいるので、この特徴と繰り返し構文を利用すれば、簡単に配列のすべてのデータを扱うことができます。

繰り返し処理に使うfor文を使って、配列の全データを参照してみましょう。

【4-1_sample2.c】

```
#include <stdio.h>

int main() {
    int data[3] = { 90, 75, 80 };
    int i;

    for(i = 0; i < 3; i++) {
        printf("data[%d]:%d ", i, data[i]);
    }
    return 0;
}
```

▼

```
data[0]:90 data[1]:75 data[2]:80
```

このように、for文を使ってすべてのデータを参照することができました。

3 添え字の範囲

配列で「前から何番目のデータ」を表すのには、添え字番号を使います。

```
int d[3]*4 = { 90, 75, 80 };
```

ヒント

*4：配列定義時の添え字は、配列の大きさを意味します。

この配列の場合、データの数は3つです。最後のデータを表す添え字番号は「2」なので、最後のデータを参照するには「d[2]」と書きます。つまり、配列の最大添え字番号は、「定

義したデータの数 － 1」です。添え字で利用できる数値は、「0 ～（定義したデータの数 － 1)」になります。

では、それ以外の数値を添え字に指定した場合はどうなるでしょうか。

```
d[10] = 100;
printf("%d", d[-1]);
```

例えば上記のように指定した場合、コンパイラによってはエラーになりませんが、予期しない数値が表示されることがあります。添え字の範囲には十分注意しましょう。

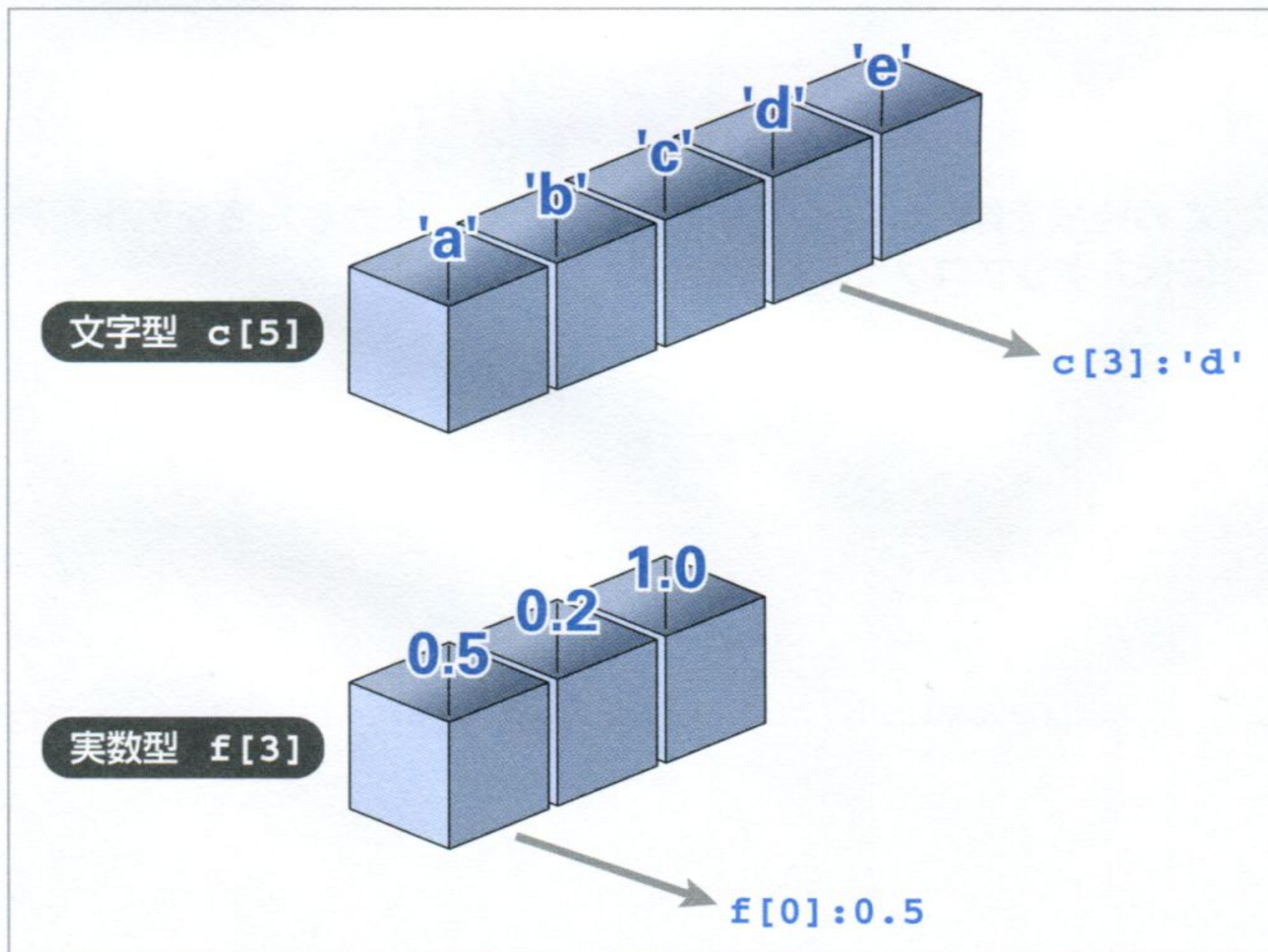
4 その他のデータ型配列

整数型の他に、文字型、実数型の配列を作ることができます。

整数型と同じく、変数を定義するときに格納するデータの型を宣言します。

【文字型、実数型の配列定義】

```
char c[5] = { 'a', 'b', 'c', 'd', 'e' };  ←文字型

float f[3] = { 0.5, 0.2, 1.0 };  ←実数型
```

●文字型、実数型の配列を定義する

それぞれのデータの参照方法も、整数型のときと同様です。

注意しなければならないのは、配列は同じ型のデータのあつまりなので、

```
int d[3] = { 90, 65.2, 80 };
```

と、違う型のデータを定義してはいけないということです。gccコンパイラではエラーにはなりませんが、d[1]を参照すると、65.2は整数に丸め込まれて65となってしまいます。

5 10個の入力値を保存する

データを10回入力し、それをすべて保存してあとから表示するプログラムでは、まずデータ格納用に配列を用意します。

```
int kotae[10];
```

ヒント
*5：数値以外の値が入力された場合は、0を設定します。

for文により10回繰り返し入力を行い、その入力値をこの配列に格納します[*5]。その値を順に表示すれば、プログラムの完成です。

まとめ

配列の基礎を学習しました。配列とfor文を使うと、データの扱いがとても楽になります。この2つの使い方はセットでおぼえましょう。これからもよく出てきます。

また、添え字番号は0から数えはじめることが重要です。

練習問題

Q

この時限で作成した4-1.cプログラムを、入力した値を直接配列に代入するプログラムに改造しなさい。

解答は巻末に

COLUMN

配列の初期値

int型変数で初期値を設定しないと、予期しない値が入っています。
ではint型配列の場合はどうでしょうか。

```
int d[3];
```

d[0]～d[2]までの値を出力してみると、やはりすべての要素に予期しない値が入っています。
すべての要素の初期値に0を設定したい場合は、

```
int d[3] = {0, 0, 0};
```

と初期値を設定すればよいのですが、次のような方法を使うこともできます。

```
int d[3] = { 0 };
```

これで配列の全要素が0で初期化されました。0以外の他の値ではできません。試しに{1}を初期値とすると、{1, 0, 0}で初期値が設定されます。
全要素を0で初期化するときは、この方法を使いましょう。

また、変数はmain関数の外に宣言することもできます。これはグローバル変数といって第7日で学習する予定です。変数としての役目はmain関数の中に宣言したものとほぼ同じですが、グローバル変数の場合、初期値を設定しなくてもすべての要素が0で初期化されます。

他のデータ型配列で初期値を設定しない場合の値を見てみると、実数型配列の場合は全要素が0.0で初期化されています。これはグローバル変数であってもなくても同じです。
文字型配列の場合、グローバル変数の場合は全要素が'¥0'で初期化されますが、main関数の中に宣言した場合（ローカル変数といいます）は、'¥0'ではなく予期しない値が入ってます。

配列の初期値を設定しなかった場合、それぞれの要素に設定されている値は、変数を宣言する場所とデータ型によって異なります。

第4日 2時限目

脳トレゲームⅠを作ろう

【脳トレゲームを作ろう②】

10問中、足し算と引き算を半分ずつ出題する脳トレゲームを作ります。

今回作成する例題

```
C:¥source>noutore1.exe
 9 +  5 = 14
 5 + 12 = 17
10 + 12 = 22
 9 + 12 = 21
15 + 15 = 30
15 -  5 = 10
 3 -  1 = 2
37 - 19 = 18
29 - 14 = 15
24 -  4 = 20
10 問中 10 問正解！
平均解答時間 2.137
判定は・・・優秀！
C:¥source>
```

足し算と引き算の問題を解き…

その時間から、結果を判定する

サンプルファイルはこちら 10days_c ▶ day04-02 ▶ noutore1.c

●このレッスンのねらい

1時限目では、配列の基本について学びました。脳トレゲームプログラムでは、3つの配列を利用してゲームを作ります。
また、ここでは時間経過を測定する方法についても学習し、出題された計算問題を解いて答えるまでの時間を計測して、判定材料とします。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int kotae[10];  //解答を入れる配列
    int seikai[10]; //正解を入れる配列
    double jikan[10];    //解答入力時間を入れる配列
    int i;
    int j, k;
    int r;               //入力した答え
    clock_t start,end;
    int kekka = 0;       //正解の回数
    double kekka_jikan = 0;

    srand(time(NULL));
    for(i = 0; i < 5; i++) {
        r = -1;
        j = rand()%20 + 1;
        k = rand()%20 + 1;
        printf("%2d + %2d = ", j, k);
        start = clock();
        scanf("%d", &r);
        while (getchar() != '¥n') {  }
        end = clock();
        kotae[i] = r;
        seikai[i] = j + k;
        jikan[i] = (double)(end-start) / CLOCKS_PER_SEC;
    }
    for(i = 5; i < 10; i++) {
        r = -1;
        j = rand()%20 + 1;
        k = rand()%20 + 1;
        printf("%2d - %2d = ", j+k, j);
        start = clock();
        scanf("%d", &r);
        while (getchar() != '¥n') {  }
        end = clock();
```

```
        kotae[i] = r;
        seikai[i] = k;
        jikan[i] = (double)(end-start) / CLOCKS_PER_SEC;
    }

    for(i = 0; i < 10; i++) {
        kekka_jikan += jikan[i];
        if(kotae[i] == seikai[i]) { kekka++; }
        else { kekka_jikan += 3; } // 不正解ペナルティ
    }
    printf("10 問中 %d 問正解！¥n", kekka);
    printf("平均解答時間 %.3f¥n", kekka_jikan / 10);
    printf("判定は・・・");
    if(kekka_jikan < 25) { printf("優秀！"); }
    else if(kekka_jikan < 35) { printf("普通"); }
    else { printf("遅い"); }
    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「noutore1.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、noutore1.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o noutore1 noutore1.c
```

4 プログラムを実行する。出題と解答を10回繰り返し、判定結果が表示されれば成功！

```
C:¥source>noutore1.exe
```

▼

```
17 +  3 = 20
 3 +  8 = 11
```

```
13 + 16 = 29
 7 + 16 = 23
19 + 13 = 32
21 - 14 = 7
13 -  5 = 8
26 - 12 = 14
21 - 15 = 6
28 -  8 = 20
10 問中 10 問正解！
平均解答時間 3.310
判定は・・・普通
```

解説

1 時間を測定する

ここで作成する脳トレゲームでは、計算問題を10問出題します。1問出題してから解答を入力するまでの時間を測定し、その時間を配列変数に記録しておきます。

(1) clock関数

出題から解答入力までの経過時間を測定するのには、clock関数を使います。この関数は、プログラム実行開始からの経過時間*2（プロセッサ時間）を返します。

> ヒント
> *2：経過時間の精度は、処理系に依存します。

clock関数を2回呼び出してその差を求めれば、経過時間を算出することができます。次のプログラムを実行してみてください。

【4-2_sample1.c】

```
#include <stdio.h>
#include <time.h>

int main() {
    int r = 0;
    clock_t start, end;
    double jikan;

    printf("数字を入力してください > ");
    start = clock();  ←測定開始
    scanf("%d", &r);
    end = clock();  ←測定終了
    jikan = (double)(end-start) / CLOCKS_PER_SEC;
    printf("入力までの時間は %lf 秒です", jikan);
    return 0;
}
```

```
数字を入力してください > 123
入力までの時間は 2.463000 秒です
```

測定開始時間を変数startに、終了を変数endに記録します。このstartとendはclock関数が返す値のデータ型と同じで、clock_t型です。環境にもよりますが、clock_t型はsigned long型と同じ[*3]だと思ってください。

経過時間を秒で表現するには、CLOCKS_PER_SEC[*4]というもので割る必要があります。これはマクロと呼ばれる定数で、第9日に詳しく説明します。

clock関数[*5]を使うにはtime.hをインクルードし、経過時間の差をCLOCKS_PER_SECで割るものだと、ここではおぼえておいてください。

ヒント
*3：C言語では、intやcharなどの基本データ型を使って、用途によりさまざまな別のデータ型が多数定義されています。clock_t型はそのひとつです。

ヒント
*4：CLOCKS_PER_SECの値は、C:¥mingw-jp¥include¥time.hで定義されています。

ヒント
*5：clock関数での値取得には上限があります。比較的短い時間で秒以下の単位を取得するにはclock関数を用い、長時間の計測を行うにはdifftime関数を用いてください。

(2) キャスト

経過時間の差を求めるとき、CLOCKS_PER_SECで割る他に(double)というものをつけました。

```
(double)(end-start) / CLOCKS_PER_SEC
```

もし(double)をつけないで実行すると、

```
入力までの時間は 2.000000 秒です
```

結果はこのようになり、小数点以下が出てきません。(double)をつけることにより、(end-start) / CLOCKS_PER_SECの計算結果がdouble型に変換されます。これを、double型へのキャストと呼びます。

キャストとは、一時的にデータ型を変換するためのものです。キャストによるデータ型変換は、他のデータ型でも可能です。

2 脳トレゲームを作る

では、脳トレゲームのポイントを見ていきましょう。

(1) 脳トレゲームの流れ

脳トレゲームプログラムで使用する配列は「解答」「正解」「測定時間」の3つで、それぞれ10問分のデータを格納して、あとからこの3つの配列値を使って答えあわせを行います。

```
int kotae[10];   // 解答を入れる配列
int seikai[10]; // 正解を入れる配列
double jikan[10];    // 解答入力時間を入れる配列
```

解答入力までの時間を入れる「測定時間」の配列のみ、double型になります。

ここで作成する脳トレゲームの流れは、次のようになります。

●脳トレゲームⅠの流れ

```
for(10問分繰り返す) {
    問題を作る
    問題を出す（時間測定開始）
    解答を読み込む（時間測定終了）
    解答と正解、および時間をそれぞれ配列に記録
}
for(10問分繰り返す) {
    解答までの時間を合計する
    正解の回数を数える（不正解の場合は時間ペナルティ追加）
}
結果を発表する
    正解数と平均解答時間
    解答までの合計時間*6が25秒より少なければ「優秀」、35秒より少なければ「普通」、
    それ以上ならば「遅い」
```

ヒント

*6：ペナルティ追加時間や結果判定の時間は適当です。実際にプログラムを動かしてみて、自分で好きなように値を変更してみてください。

コードが少し長くなりますが、きちんと順序だててプログラムを書きます。

(2) 問題を作る

出題する問題は、足し算と引き算を5問ずつ作成します。
足し算は1～20までのランダムな数値を2つ用意し、それを足すものです。

【足し算問題】

```
j = rand()%20 + 1;
k = rand()%20 + 1;
printf("%2d + %2d = ", j, k);
seikai[i] = j + k;
```

引き算は1～20までのランダムな数値を2つ用意し、そのうちの一方が引き算の答えになるような問題にします。

【引き算問題】

```
j = rand()%20 + 1;
k = rand()%20 + 1;
printf("%2d - %2d = ", j+k, j);
seikai[i] = k;
```

(3) 脳トレゲームを完成させる

(1) で説明した脳トレゲームの流れを追って、プログラムを完成させましょう。

問題の解答を一時的に格納する変数rには、毎回、-1を代入しています。

```
r = -1;
```

解答の正解に-1が入ることはありません。もしも文字や文字列などの間違った入力が行われた場合は、正解と一致することのないように、毎回、-1を設定しておきます。

判定に使用する時間は、問題を出題してから解答を入力するまでの時間を測定したものです。clock関数を使う位置に注意しましょう。

まとめ

乱数、繰り返し、条件分岐と今まで習ったことの集大成のようなプログラムになりました。

うまく動かない場合は、プログラムの一部分だけが実行するように他の部分をコメントにして、問題の箇所を探しましょう。

練習問題

Q

掛け算、割り算の問題を、以下の条件で作成しなさい。

・掛け算：(2～20までの数値) × (2～9までの数値)
・割り算：余りの出ない計算で、割る数2～20、答えは2～9までの数値

解答は巻末に

COLUMN

時間を計測する

この時限のプログラムでは時間の計測にclock関数を使用しましたが、長時間の計測を行うにはdifftime関数を使用します。
ジャンケンゲームプログラムでランダムな値を算出するのに使用したsrand関数を思い出してください。srand関数の引数には現在時刻を表す数値を使用しました。これはtime関数を使って算出します。
time(NULL)で返される値のデータ型はint型ではなく、time_t型という時刻を表す型です。詳しくは第８日に説明するので、今は単にint型のような型と思ってください。
とある２地点で、そのときの時間をtime関数で計測しておき、その値をdifftime関数の引数にして時間の差を求めます。

```
double difftime(time_t t1, time_t t0);
```

t1が計測終了時間、t0が計測開始時間です。戻り値はdouble型になります。
difftime関数とtime関数を使う場合、time.hをインクルードします。
実際の測定方法は次のとおりです。

```
time_t t0, t1;
t0 = time(NULL);
          (時間経過)
t1 = time(NULL);
printf("%d 秒" , (int)difftime(t1, t0));
```

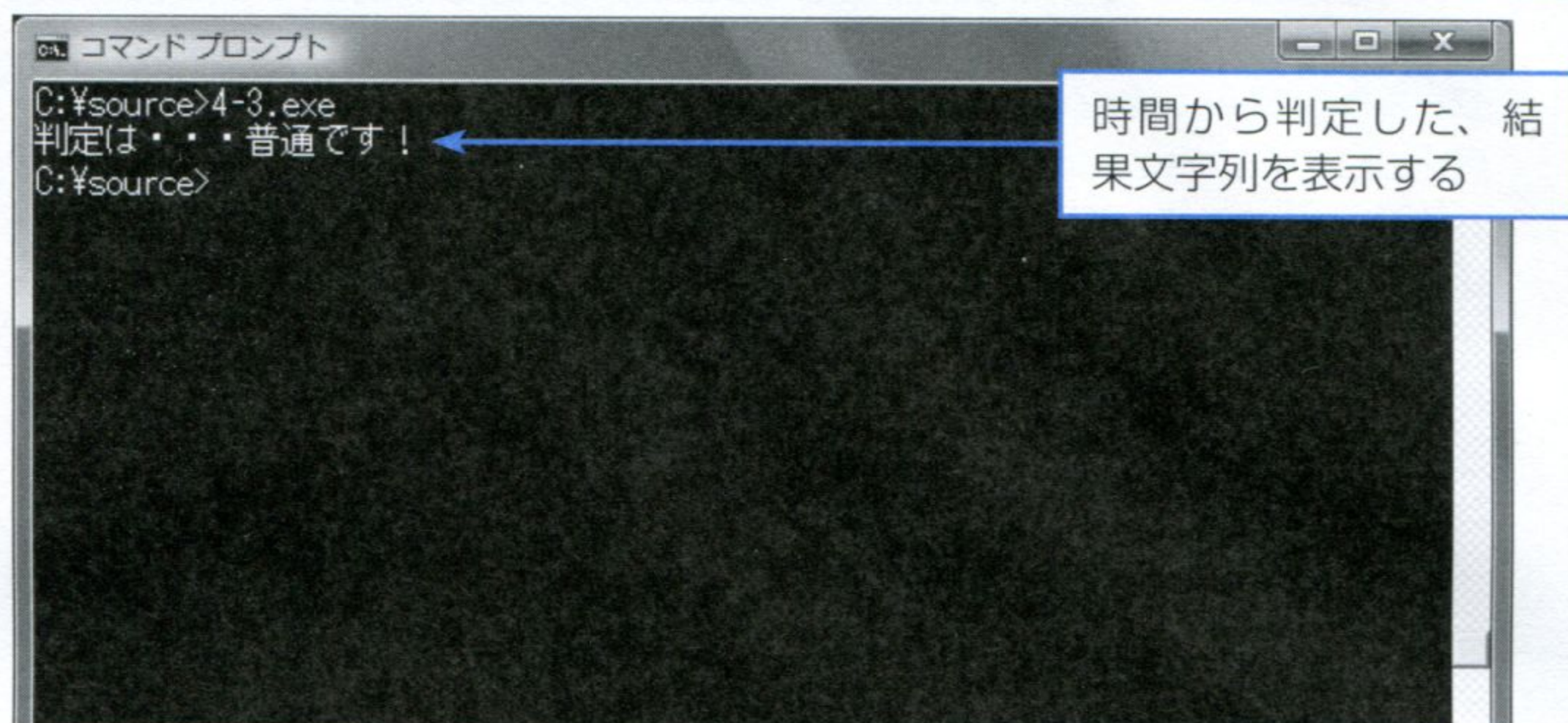
時間経過が１秒に満たない場合は、０秒となります。
短時間でコンマ以降の値を計測したい場合は、この時限で学習したようにclock関数を使いましょう。

第4日 3時限目 【脳トレゲームを作ろう③】文字列と配列について学ぼう

ここでは、特殊な文字型配列である文字列について学習します。

今回作成する例題

```
C:¥source>4-3.exe
判定は・・・普通です！
C:¥source>
```

時間から判定した、結果文字列を表示する

サンプルファイルはこちら 10days_c ▶ day04-03 ▶ 4-3.c

●このレッスンのねらい

本日の１時限目では、配列の基本について学びました。
Ｃ言語では文字型のデータのあつまり（配列）を少し工夫して、文字列として扱います。これは、普通の文字型配列とは少しだけ異なります。
本レッスンでは文字列の基礎と、文字列を扱う関数について学習しましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <string.h>

int main() {
    double kekka_jikan = 30.0;
    char kekka_str[10];

    if(kekka_jikan < 25) { strcpy(kekka_str,"優秀！"); }
    else if(kekka_jikan < 35) { strcpy(kekka_str,"普通"); }
    else { strcpy(kekka_str,"遅い"); }
    printf("判定は・・・%sです！", kekka_str);
    return 0;
}
```

2 入力できたら、「4-3.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント

*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、4-3.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 4-3 4-3.c
```

4 プログラムを実行する。「判定は・・・普通です！」と表示されれば成功！

```
C:¥source>4-3.exe
```

```
判定は・・・普通です！
```

解説

1 文字列

プログラムでは、さまざまな型のデータを扱うことができます。データはそれぞれ変数に保存して、参照や変更を行います。整数の場合はint型の変数、文字の場合はchar型の変数に保存します。では、文字列を変数に保存する場合はどうしたらよいのでしょう？

(1) 文字列の基本

数値や文字を変数に保存するには、次のようにしました。

```
int d = 10;
char c = 'a';
```

では、文字列の場合は？

文字列は、読んで字のごとく「文字」の「列」、つまり文字のあつまりです。文字のあつまりということは、文字型の配列（文字型配列）を使えば、文字列を変数に保存できます。

文字列「abc」を表す場合は配列を使って、

【文字型配列の定義①】

```
char str[] = "abc";
```

と書きます。または、

【文字型配列の定義②】

```
char str[4] = "abc";
```

と書きます。文字列はダブルクォート「"」で括ります。

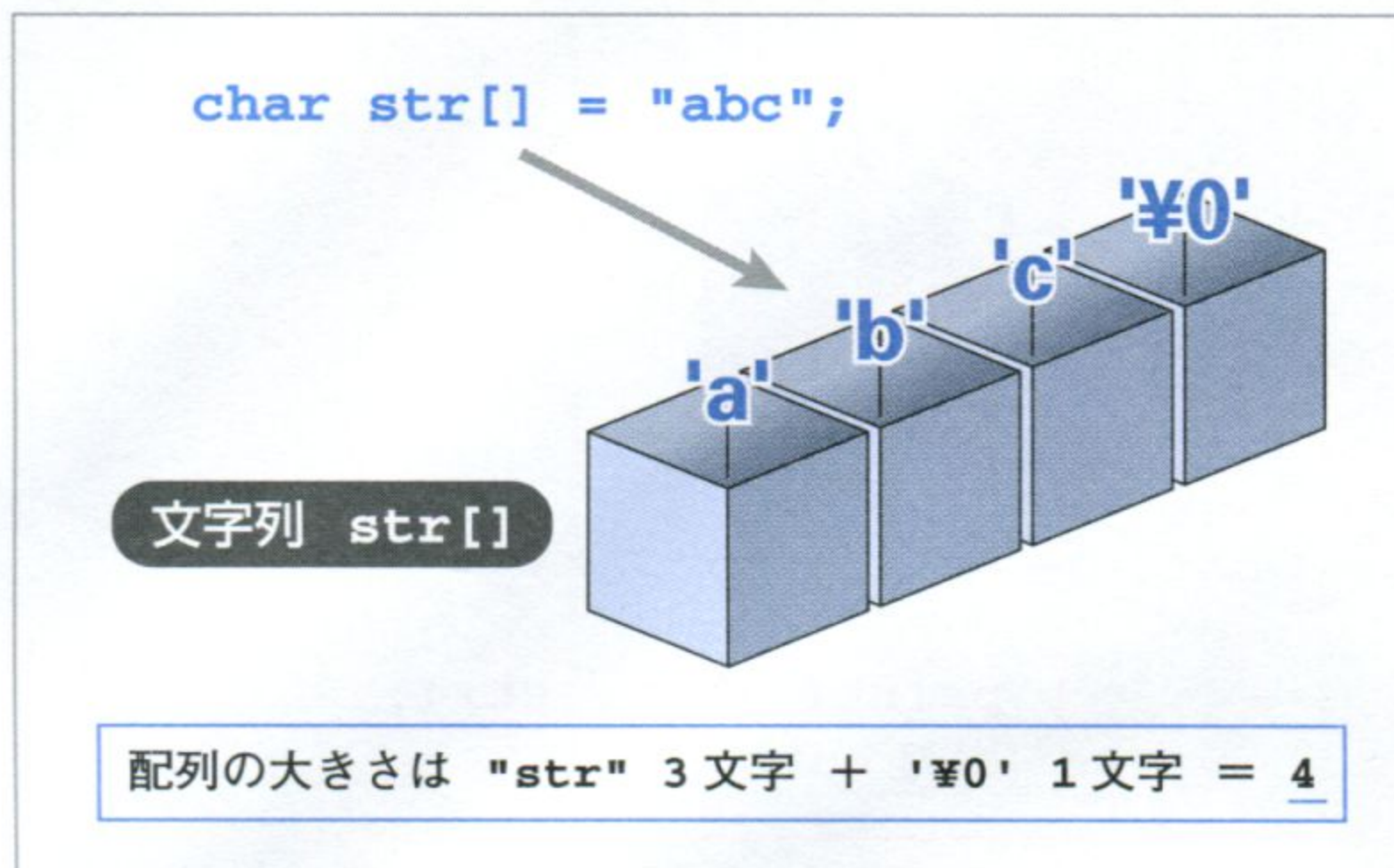
ここで、「おや？」と思った方もいるでしょう。配列名の後の[ ]の中には、配列要素の個数を書きます。「abc」は「a」「b」「c」という3つの文字型データのあつまりなのだから、【文字型配列の定義②】に場合は、str[3]と書きたいところです。しかし、C言語では文字列の最後には必ず「¥0」が来るきまりになっています。つまり、

```
char str[] = "abc";
```

と定義すると、文字列の最後を表す「¥0*2」が自動的に最後の文字の後に格納されます。

ヒント

*2：「¥0」は1文字として扱います。

●文字列の大きさ

よって、配列strの大きさは4になります。文字列は文字型データの配列ですから、

【文字型配列の定義③】

```
char str[4] = { 'a', 'b', 'c', '¥0' };
```

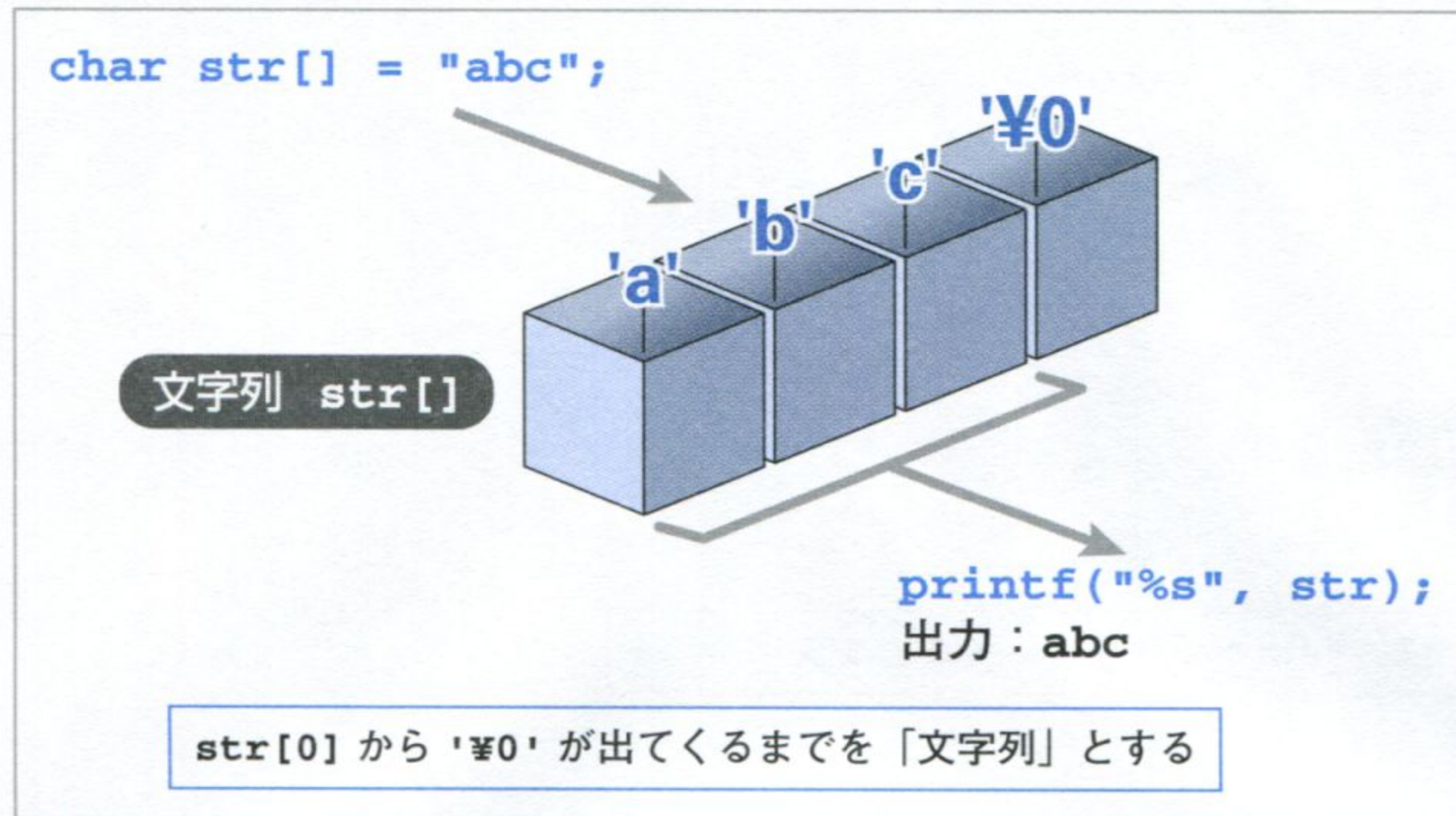
と書いても同じです。この場合は、最後に「¥0」を書くことを忘れないでください。

(2) 文字列は特殊な文字型配列

文字列は「文字型データのあつまり」、つまり文字型配列と説明しましたが、正確には特殊な文字型配列です。配列の最後に「¥0」を定義せずに、

```
char str[] = { 'a', 'b', 'c' };
```

と書いた場合、これはあくまで文字型の配列であり、まとめて「文字列」として扱うことはできません。文字列を表す処理では、配列の先頭文字から「¥0」が出てくるまでを「文字列」として扱います。

●文字列は配列の先頭文字から¥0まで

「¥0」が最後にない文字型配列を文字列として扱おうとすると、予期しない動作をします。次のプログラムを実行してみましょう。

【4-3_sample1.c】

```
#include <stdio.h>

int main() {
    char str1[] = "abc";
    char str2[] = { 'a', 'b', 'c'};

    printf("str1:%s ¥nstr2:%s¥n", str1, str2);
    return 0;
}
```

▼

```
str1:abc
str2:abc
```

文字列の出力書式は「%s」です。変数str1は最後が「¥0」なので「文字列」として扱いますが、変数str2はただの文字型配列のため、文字列の書式で出力しても「abc」とは出力されません。環境にもよりますが、おそらく妙な表示になるでしょう。

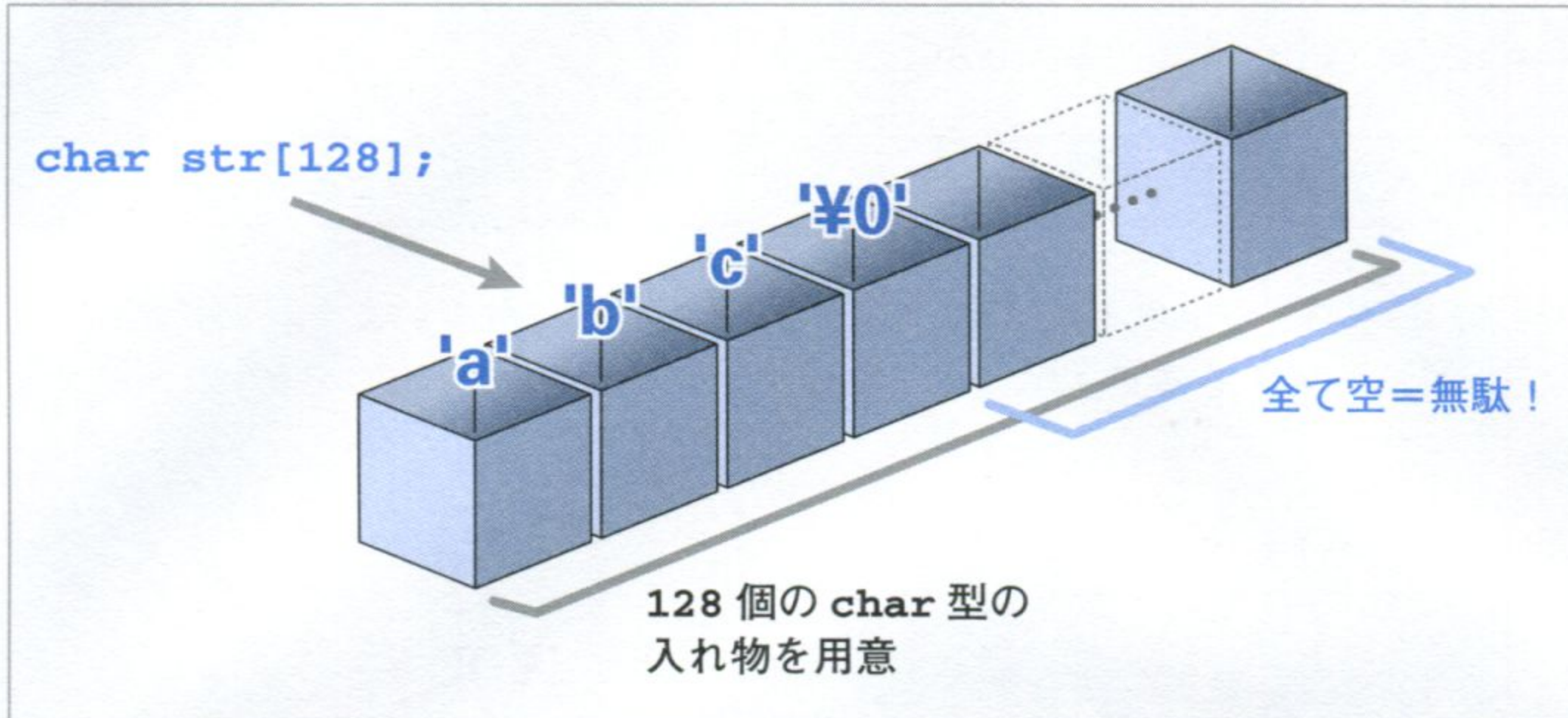
文字列は、「¥0」が出てくるまで文字列とみなされます。このため、

```
char str[128];
```

と定義した場合、127文字を絶対に入れなければならないわけではありません。

```
char str[128] = "abc";
```

と初期値を設定すると、配列の最初の4つだけに値が入ることになります。

●配列の大きさは適切に！

配列の残りには何も入っていません。何も入っていなくても、入れ物だけは128個分用意されている状態です。これは無駄です。

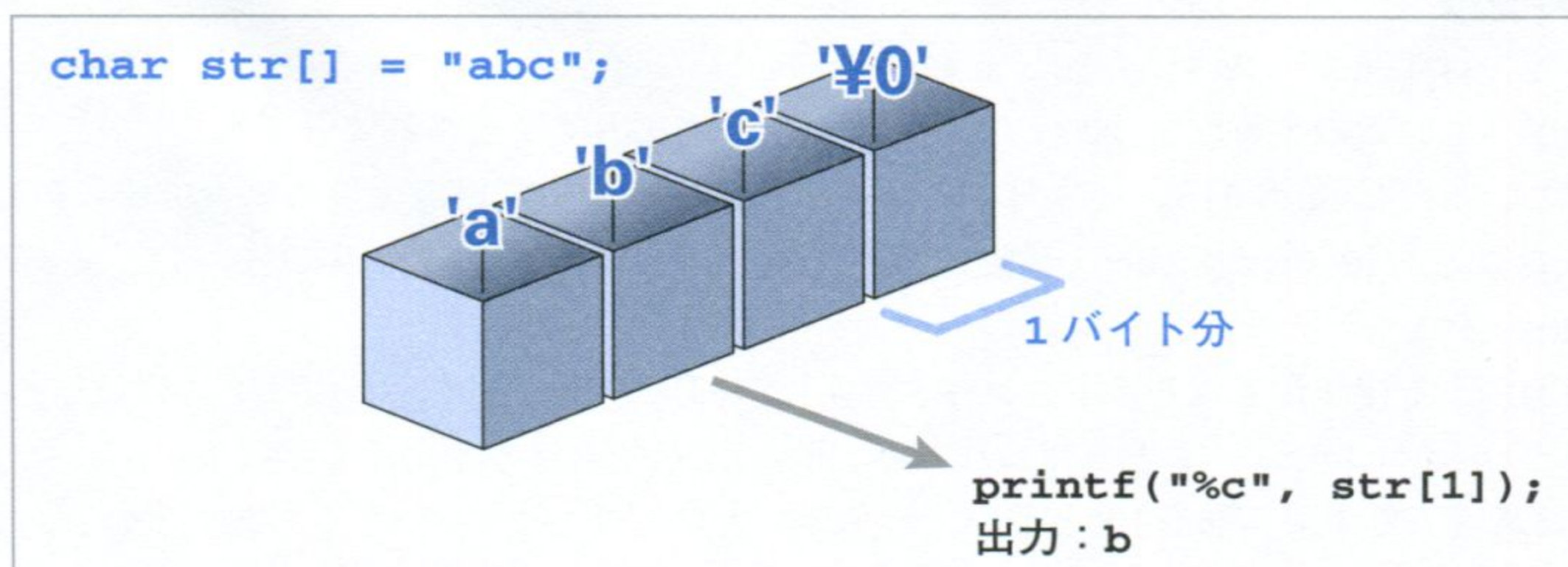
配列の大きさは必要最低限のものに設定しましょう。

2 日本語文字列の扱い

文字列とは、特殊な文字型データの配列です。配列のひとつひとつには、それぞれ文字型データがひとつずつ格納されています。

(1) ASCII文字列と配列の長さ

文字型データの大きさは、1バイトです。文字「a」「b」「c」はASCII文字でそれぞれ1バイトなので、配列の各要素にちょうどおさまっています。「¥0」も特殊文字ですが、これも1バイト分です。ASCII文字*3だけで構成された文字列の配列の長さは、「文字数 + 1」になります。

●ASCII文字列から1文字抜き出す

ヒント

*3：ASCII文字とは、1963年にアメリカ規格協会（ANSI）が定めた、情報交換用の文字コードの体系です。半角英数文字や記号文字をASCII文字といいます。

str[添え字番号]と書くと、配列の各要素を1文字ずつ別に取り出すことができます。

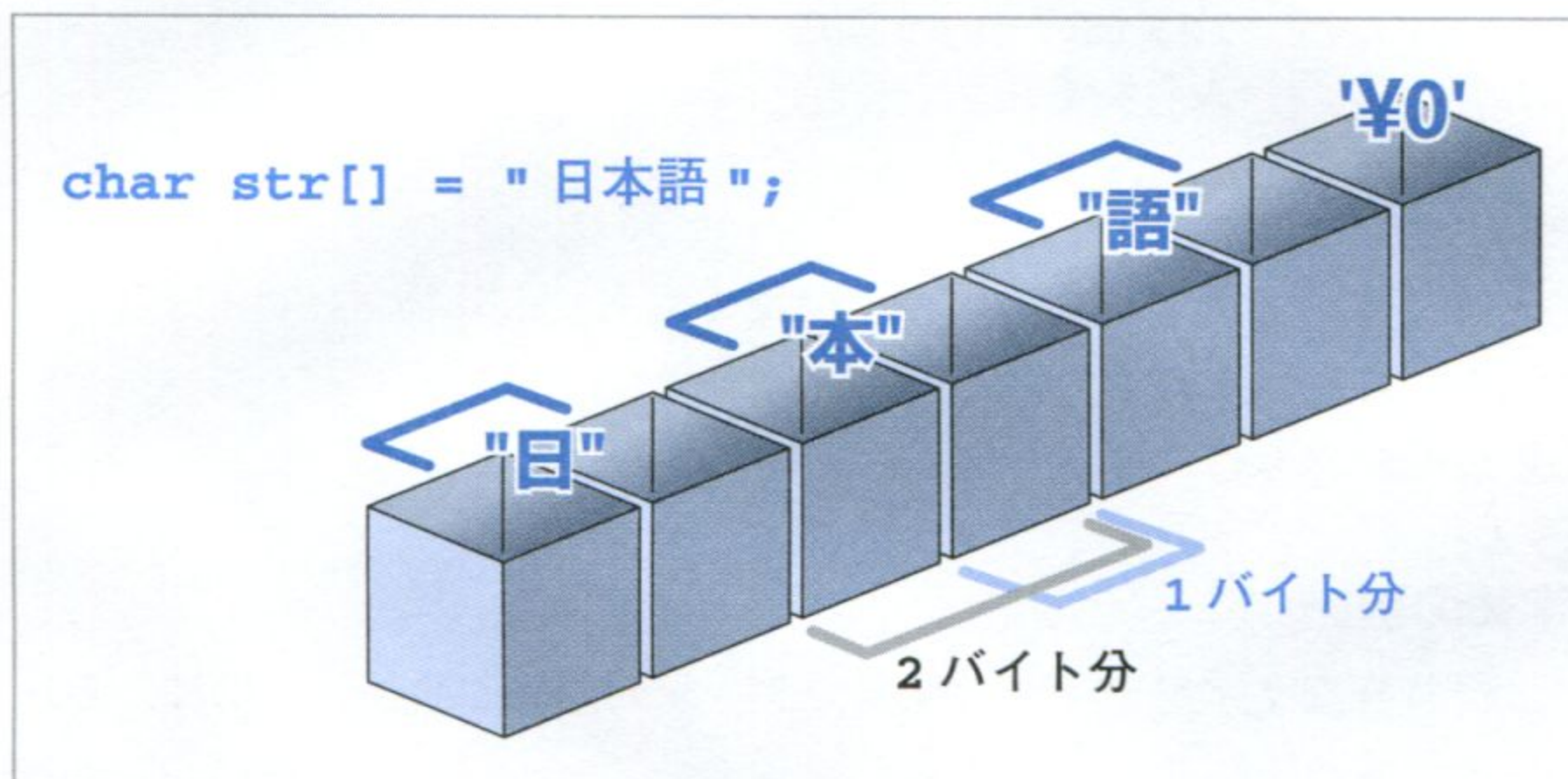
なお、ここまでの説明は、文字列に格納するデータがすべてASCII文字の場合です。

(2) 日本語文字列

文字列では日本語も扱えます。定義方法はASCII文字のみの場合と同じです。では、次の配列の長さはいくつになるでしょうか。

```
char str[] = " 日本語 ";
```

基本的に、日本語（の全角文字）は1文字2バイトで構成されています。文字型配列のひとつの要素は1バイトなので、2つ分を使って、やっと日本語の1文字を表すことができます。

●日本語は1文字2バイト

つまり最初の「日」は、str[0]とstr[1]の2つにまたがって格納されます。2つでひとつの文字になるので、その一方だけのstr[1]を出力表示しようとしても、なにやらワケのわからないものが出力されてしまいます。日本語の文字列の扱いには注意しましょう。

ASCII文字1文字は1バイトですが、全角1文字は2バイトで数えます。文字列の最後には必ず「¥0」がつくので、配列strの長さは、7[*4]（全角3文字×2 + 1）になります。

(3) 半角カナ

日本語には半角カナ文字が存在します。「ｱｲｳｴｵ」などが半角カナ文字です。同じ日本語の文字でも、この半角カナは2バイトではなく、1バイトで構成されています

ただし、これはShift-JISという日本語コード体系を採用している環境の場合です。PCに代表されるWindowsやMac、最近では携帯電話で使われている文字コードは、このShift-JISです。

この他によく使われている日本語コード体系に、EUCがあります。これは主にUNIXと呼ばれるOSのコンピュータ上で使われています。EUCでも日本語全角文字はShift-JISと同じく2バイトですが、Shift-JISと違い、半角カナ文字もひとつで2バイト必要です。どちらにしろ、半角カナ文字はプログラムでは使わないようにしましょう。

ヒント

*4：これは日本語の文字コードにShift-JISを使用している場合です。

③ 文字列の入出力

文字列の入出力は、文字や数値の場合と同様にできます。

文字列を扱う書式は「%s」です。ただし、文字列の場合はscanf関数の第2引数の前に、「&」をつける必要はありません。

入力した文字列をそのまま出力するプログラムを作り、半角で63文字以内の文字列を入力してみましょう。

【4-3_sample2.c】

```
#include <stdio.h>

int main() {
    char str[64];

    printf(" 文字列を入力して下さい > ");
    scanf("%s", str);          ←── & はつけない
    printf(" 入力文字列：%s", str);
    return 0;
}
```

▼

```
文字列を入力して下さい > 文字列！
入力文字列：文字列！
```

ASCII文字も日本語文字も、扱うことができます。

④ 文字列関数

文字列を扱う場合、これまでは文字列を定義するときに、一緒に初期値を設定しました。しかし、あとで値を設定したり、設定した値を変更したりしたい場合は、どうしたらよいでしょうか。

文字列は文字型データの配列ですから、配列の各データに値をそれぞれ入力します。文字列を表す「¥0」を、最後に忘れずに定義します。

【文字型配列の定義④】

```
char str[4];

str[0] = 'a';
str[1] = 'b';
str[2] = 'c';
str[3] = '¥0';
```

データを変更する場合も同様です。

例えば文字列を短くしたいときは、「¥0」を代入することで、文字列はそこまでの長さになります。次のように書くと、文字列strは「ab」になります。

```
char str[] = "abc";
str[2] = '¥0';
```

ASCII文字列を1文字だけ変更する場合は、この方法で問題ありません。しかし、日本語全角文字は2バイトなので、この方法が使えません。また、ASCII文字のみでも文字数が多ければ多いほど、1文字ずつ設定するのは面倒です。C言語では、文字列をコピーしたりつなげたりする文字列関数が用意されているので、それを使いましょう。

なお、ここで紹介する文字列関数を使うためには、string.hファイルをインクルードする必要があります。ファイルの最初に、

```
#include <sring.h>
```

と忘れずに書きましょう。

（1）文字列の長さを取得するstrlen関数

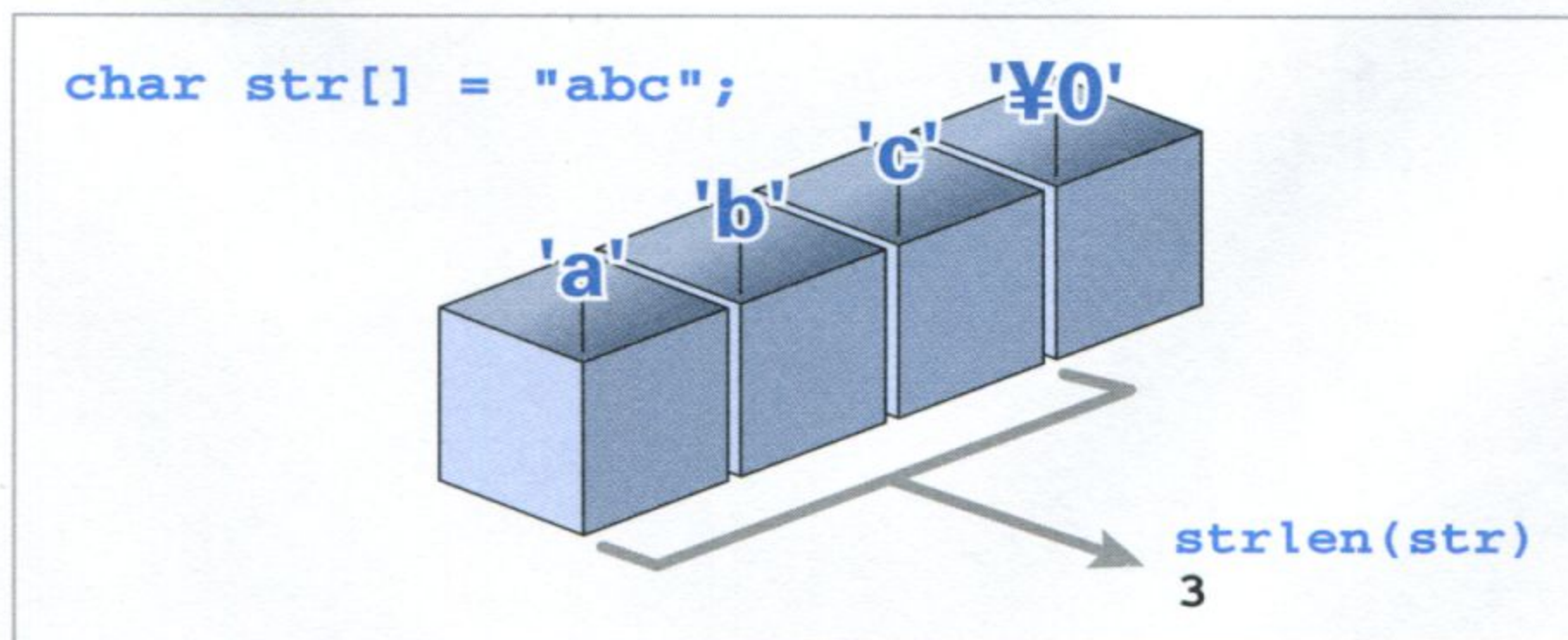
文字列の長さを求めるには、strlen関数を使います。長さを求める文字列を引数に指定して、

【strlen関数】

```
strlen( 文字列 )
```

と呼び出します。すると、指定した文字列の長さを返します。

strlen関数で取得できる結果は、文字列の最後の「¥0」を含めない長さになります。つまり、strlen関数は「配列の大きさ − 1」の値を返します。

●strlen関数は最後の¥0は数えない

簡単なサンプルで、strlen関数の動作を確認しましょう。

【4-3_sample3.c】

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "abcde";
    char jstr[] = "日本語文字列";

    printf("%s length=%d¥n", str, strlen(str));
    printf("%s length=%d¥n", jstr, strlen(jstr));
    return 0;
}
```

```
abcde length=5
日本語文字列 length=12
```

(2) 文字列をコピーするstrcpy関数

文字列変数に文字列を代入（コピー）するには、strcpy関数を使います。

【strcpy関数】

```
strcpy( 変数名 , 文字列 )
```

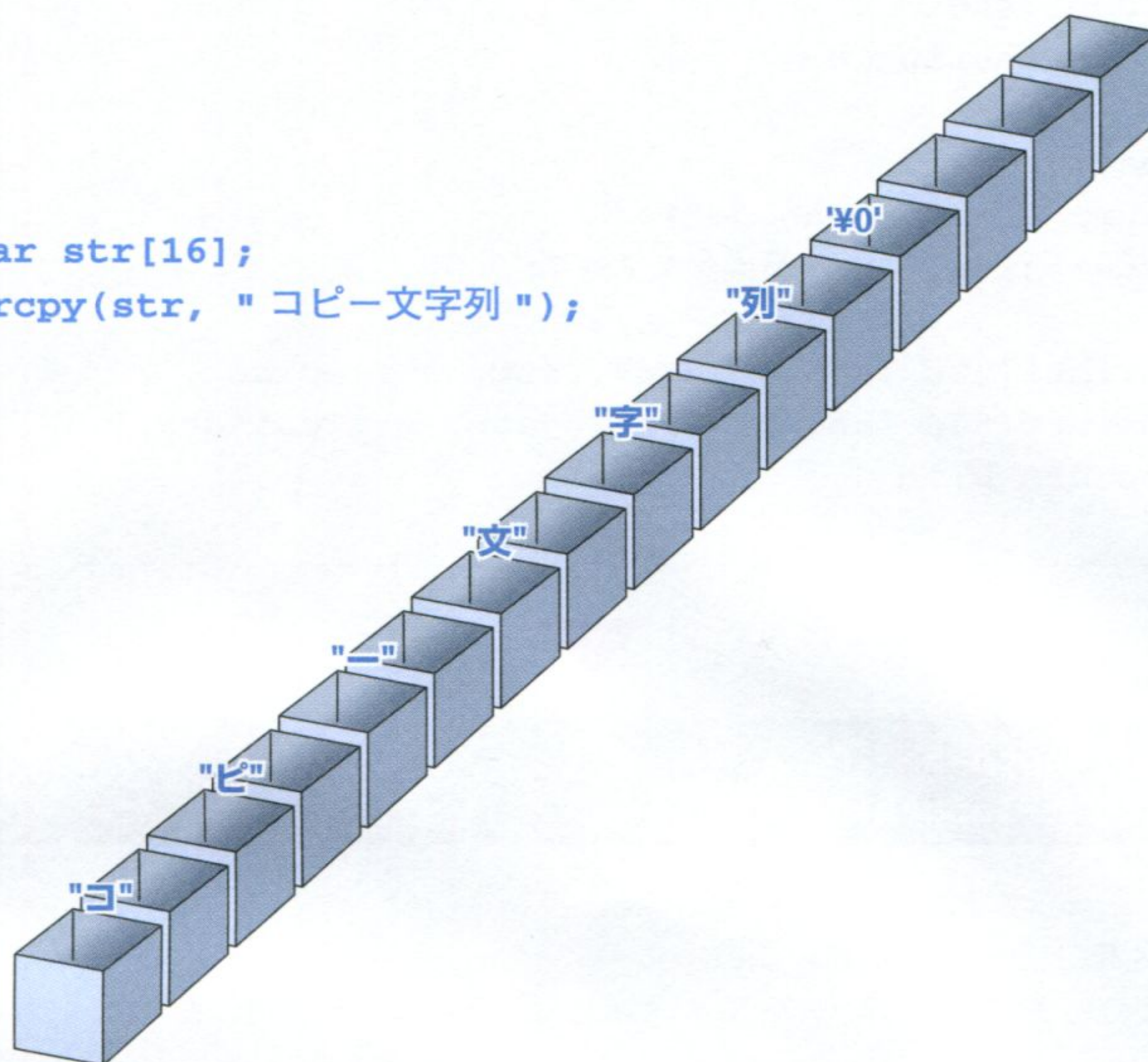
第2引数に指定した文字列*5を、第1引数の変数にコピーします。変数とは、この場合は文字列型変数になります。

```
char str[16];
strcpy(str, "コピー文字列");
```

ヒント

*5：第２引数である文字列は、変数でもかまいません。
strcpy(str, str2);

●strcpy関数で文字列変数に文字列を代入

```
char str[16];
strcpy(str, " コピー文字列 ");
```


このように書くと、文字列変数strに、文字列「コピー文字列」が代入されます。

(3) 文字列同士をつなげるstrcat関数

文字列と文字列をつなげてひとつの文字列を作り出すには、strcat関数を使います。第1引数に文字列変数を、第2引数につなげたい文字列を指定します。

【strcat関数】

```
strcat( 文字列変数 , 文字列)
```

第2引数の文字列[*6]は、第1引数の文字列変数につながります。

```
char str[8] = "abc";
strcat(str, "def");
```

ヒント
*6：第2変数である文字列は、変数でもかまいません。
strcpy(str, str2);

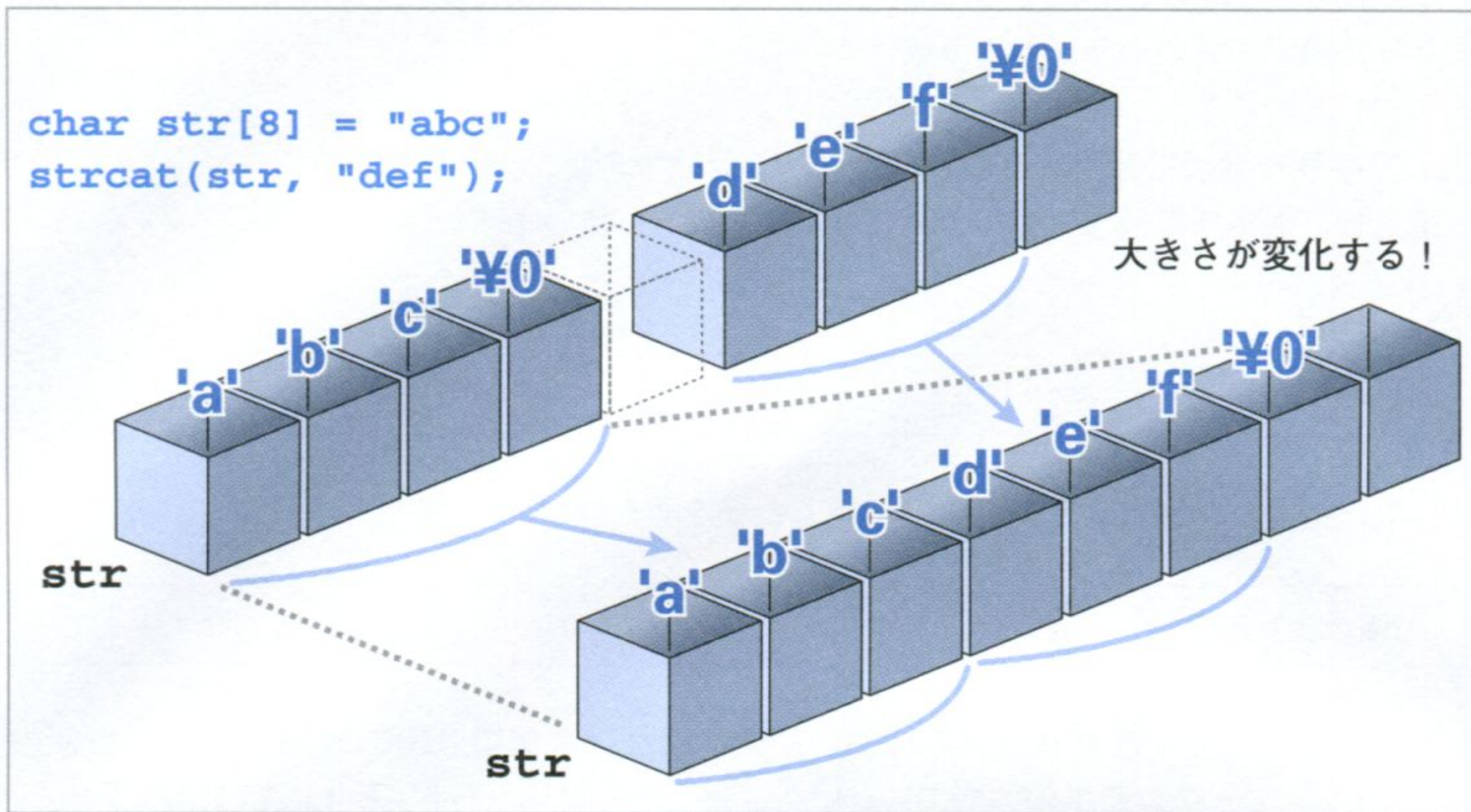
●strcat関数で文字を連結

```
char str[8] = "abc";
strcat(str, "def");
```

'a' 'b' 'c' '¥0' 'd' 'e' 'f' '¥0'

大きさが変化する！

str

str

strは「abcdef」になりました。第1引数であるstrの大きさは、strcat関数により長くなります。文字列strの文字数は7[*7]になるので、それ以上の大きさを指定しておきます。

ヒント
*7：「abc」＋「def」＋「¥0」で、7文字になります。

(4) 文字列を比較するstrcmp関数

文字列を比較するには、strcmp関数を使います。

文字列1と文字列2を比較し、比較した結果は表のようになります。

【strcmp関数】

```
strcmp( 文字列 1, 文字列 2)
```

●strcmp関数の戻り値

結果	意味
0	文字列１と文字列２は等しい
0より大きい	文字列１は文字列２より辞書順で大きい
0より小さい	文字列１は文字列２より辞書順で小さい

「辞書順で大きい」「辞書順で小さい」とは、辞書順に並べたときに、あとに来るか先に来るか、という意味です。「abc」は「agc」よりも前なので、「abc」は「agc」よりも小さいことになります。また、「4」と「12」では「12」の方が先に来るので、「12」より「4」の方が大きい[*8]ことになります。

ヒント
*8:「4」と「12」は、数値で見れば12の方が大きくなります。

```
int i;
char str1[] = "abc";
char str2[] = "def";
i = strcmp(str1, str2);
```

このように書くと、変数iには-1が代入されます。-1は0より小さいので、「abc」は「def」より辞書順で小さいことがわかります。

(5) 文字列を数値に変換するatoi関数

指定した文字列を数値に変換するには、atoi関数を使います。

【atoi関数】

```
atoi( 文字列 )
```

引数に指定した文字列を、数値に変換した値を返します。

```
char str[] = "123";
```

上記の場合、文字列変数strは「123」です。これは数値の123ではないので、(数値の123とみなして)それに1をプラスした値を取得したくても、「str + 1」とはできません。このようなとき、atoi関数を使います。atoi関数を使うためにはstring.hではなく、stdlib.hをインクルードします。

```
int i;
char str[] = "123";
i = atoi(str)+1;
```

上記の場合、変数iの値は数値の124になります。

(6) 数値を文字列に変換するsprintf関数

atoi関数の逆、つまり数値を文字列に変換するにはsprintf関数を使います。

【sprintf関数】

```
sprintf( 文字列変数 , 書式 , 数値 )
```

数値を書式に従って変換し、文字列変数に格納します。わかりにくいので実際に使ってみましょう。

sprintf関数を使うためには、string.hではなくstdio.hをインクルードします。

```
char str[128];
sprintf(str, "文字列 %d", 100);
```

文字列strには「文字列 100」が入ります。つまり、printf関数で出力する内容を標準出力ではなく、指定した文字列変数に格納するだけです。

5 文字列を作って表示する

脳トレゲームを作っているのだから脳年齢を判定したいところですが、私達は専門家ではないため、判定値は適当にきめましょう。解答までの合計時間から適当に判断して、結果を「優秀」「普通」「遅い」の文字列で表すことにします。

判定結果文字列を格納する変数を、それぞれ定義します。

```
double kekka_jikan = 30.0;   //解答までの時間
char kekka_str[10];          //判定結果の文字列

//kekka_jikanの値によりkekka_strの値を設定する処理を行う
```

この変数kekka_strに文字列をコピーします。なお、この時限で作るプログラムでは、解答までの時間は固定にしておきます。

解答までの時間により、変数kekka_strに「優秀！」「普通」「遅い」をそれぞれ代入[*9]します。最後にそれを表示して終了です。

ヒント

*9：文字列のコピーをするstrcpy関数を使うので、最初にstring.hを忘れずにインクルードしましょう。

まとめ

文字列が特殊な文字型配列であることが理解できたでしょうか。

今回のプログラムでは、文字列関数としてstrcpy関数のみ使用しましたが、今後の学習では他の文字列関数を扱うプログラムが出てきます。忘れてしまった場合は、この時限に戻って復習しましょう。

練習問題

Q

入力した文字列を逆から読んだ文字列を作成し、それを表示するプログラムを作成しなさい。入力文字列は英数半角のみ、19文字以内とする。

解答は巻末に

0～9までの数字のうち、ひとつだけ抜けている数値を当てるゲームで答えるまでの時間を計測し、結果を表示するプログラムを作ります。

今回作成する例題

```
C:¥source>noutore2.exe
932487061 => 5
247651930 => 8
437258019 => 6
780254369 => 1
903175826 => 4
571624803 => 9
468359172 => 0
371946508 => 2
138429065 => 7
509476381 => 2
10 問中 10 問正解！
平均解答時間 5.028
判定は・・・普通です！
C:¥source>
```

数値当てゲームで解答までの時間を計測し、その結果を表示する

サンプルファイルはこちら　10days_c ▶ day04-04 ▶ noutore2.c

●このレッスンのねらい

もうひとつの脳トレゲームを作成します。
出題から解答までの時間で結果を判定し、それを文字列として表示します。
この時限で作成するプログラムでは、特に目新しい項目はありません。だいぶプログラムを書くのに慣れてきた頃だと思いますので、できれば最初は自力で考えて作成してみましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

int main() {
    int kotae[10];          // 解答を入れる配列
    int seikai[10];         // 正解を入れる配列
    int tmp[9];
    double jikan[10];       // 解答入力時間を入れる配列
    int i;
    int j, k, c, m;
    int r;
    clock_t start, end;
    int kekka = 0;          // 正解の回数
    double kekka_jikan = 0;
    char kekka_str[10];

    srand(time(NULL));
    for(i = 0; i < 10; i++) {
        seikai[i] = rand()%10;
        for(j = 0, k = 0; j < 10; j++) {
            if(seikai[i] != j) { tmp[k++] = j; }
        }
        for(j = 9; j > 1 ; j--) {
            c = rand()%j;
            printf("%d", tmp[c]);
            for(m = c; m < j-1; m++) { tmp[m] = tmp[m+1]; }
        }
        printf("%d => ", tmp[0]);

        r = -1;
        start = clock();
        scanf("%d", &r);
        while (getchar() != '¥n') {  }
        kotae[i] = r;
        end = clock();
        jikan[i] = (double)(end-start)/CLOCKS_PER_SEC;
```

```
    }
    for(i = 0; i < 10; i++) {
        kekka_jikan += jikan[i];
        if(kotae[i] == seikai[i]) { kekka++; }
        else { kekka_jikan += 5; } // 不正解ペナルティ
    }
    printf("10 問中 %d 問正解！¥n", kekka);
    printf("平均解答時間 %.3f¥n", kekka_jikan/10);

    if(kekka_jikan < 40) { strcpy(kekka_str,"優秀！"); }
    else if(kekka_jikan < 60) { strcpy(kekka_str,"普通"); }
    else { strcpy(kekka_str,"遅い"); }
    printf("判定は・・・%s です！", kekka_str);

    return 0;
}
```

ヒント
*1：拡張子に注意して保存しましょう。

2 入力できたら、「noutore2.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、noutore2.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o noutore2 noutore2.c
```

4 プログラムを実行する。出題と解答を10回繰り返し、判定結果が表示されれば成功！

```
C:¥source>noutore2.exe
```

```
932487061 => 5
247651930 => 8
437258019 => 6
780254369 => 1
903175826 => 4
571624803 => 9
468359172 => 0
371946508 => 2
138429065 => 7
509476381 => 2
10 問中 10 問正解！
平均解答時間 5.028
判定は・・・普通です！
```

解説

1 問題を作成する

0～9の数字のうち、ひとつだけ抜けている状態を作ります。

0～9まで順番に並んでいて、ひとつだけ抜けている状態ならば、簡単に作成できます。

```
int j;
int seikai;
srand(time(NULL));

seikai = rand()%10;
for(j = 0; j < 10; j++) {
    if(seikai != j) { printf("%d", j); }
}
```

まずは抜かす数値をきめて、変数seikaiに代入します。0～9まで順に出力するうち、変数seikaiの値と一致した場合だけを抜かします。

では、0～9まで順番に並んでいない場合はどうなるでしょうか。

実現方法は色々ありますが、今回はとことん配列を利用します。まず、0～9まで順番に並んでいて、ひとつだけ抜けている状態を作ります。例えば抜かす数値が2の場合は、

```
013456789
```

このようになり、それぞれの数値をint型の配列tmpに代入します*2。

ヒント

*2：最大9種類の数値が入るので、int型配列tmpの大きさは9にします。

このint型配列tmpの中から、さらにひとつランダムに場所を選び、そこに該当する数値を出力します。例えばランダム数値が5だった場合、tmp[5]の値である6を出力します。

```
013456789    ランダム数値5→tmp[5]の値6を出力
```

配列の中から次々とランダムに場所を選び、その値を出力していけば、ランダムに並んだ数字の列を作成することができます。

しかし、この方法ではすでに出力した数値を再度選んでしまう場合があります。よって、一度出力した数値は再度選ばれることがないように、選択候補からはずします。

tmp[5]の値6を選択候補からはずすため、tmp[5]にtmp[6]の値を、tmp[6]にtmp[7]の値を……と、ひとつずつ前にずらします。

```
tmp[0] tmp[1] tmp[2] tmp[3] tmp[4] tmp[5] tmp[6] tmp[7] tmp[8]
  0      1      3      4      5             7      8      9
                              ↓
  0      1      3      4      5      7      8      9
```

次の数値出力の選択範囲はtmp[0]～tmp[7]となり、ランダム数値は0～7のどれかが選ばれるようにします。

```
01345789    ランダム数値4→tmp[4]の値5を出力
0134789     ランダム数値4→tmp[4]の値7を出力
013489      ランダム数値5→tmp[5]の値9を出力
01348       ランダム数値0→tmp[0]の値0を出力
1348        ランダム数値1→tmp[1]の値3を出力
148         ランダム数値2→tmp[2]の値8を出力
14          ランダム数値0→tmp[0]の値1を出力
4           tmp[0]の値4を出力
```

最終的に出力されるのは、

```
657903814
```

となり、最初のランダム数値2を抜かし、残りをバラバラに表示した状態になります。

これをプログラムで書くと、次のようになります。

```
int j, k, c, m;
int seikai;
int tmp[9];

srand(time(NULL));
seikai = rand()%10; printf("%d ", seikai);
for(j = 0, k = 0; j < 10; j++) {
    if(seikai != j) { tmp[k++] = j; }
}
for(j = 9; j > 1 ; j--) {
    c = rand()%j;
    printf("%d", tmp[c]);
    for(m = c; m < j-1; m++) { tmp[m] = tmp[m+1]; }
}
printf("%d", tmp[0]);
```

実際にint型配列tmpの値が短くなるわけではなく、短くなったとみなして利用しています。

2 脳トレゲームを完成させる

問題は10問なので、正解を入れる変数は配列にします。

```
int seikai[10]; // 正解を入れる配列
```

問題を10問出題します。解答までの時間は、2時限目に作成した脳トレゲームⅠと同じ方法で測定します。

```
for(i = 0; i < 10; i++) {
    seikai[i] = rand()%10;
    （問題作成）

    r = -1;
    start = clock();
    scanf("%d", &r);
    while (getchar() != '¥n') {  }
    kotae[i] = r;
    end = clock();
    jikan[i] = (double)(end-start)/CLOCKS_PER_SEC;
}
```

解答は0～9までの数値が入力されるはずですが、もしも文字など不正な入力があった場合にそなえて、解答を一時保存する変数rには、毎回、-1を代入しておきます[*3]。

さらに、結果判定は3時限目で作成した文字列のコピーを利用しましょう。

ヒント

*3：変数rに毎回0を代入しておくと、もしもその回の正解が0だった場合、不正な入力がされても正解になってしまうので、正解範囲にない数値である-1を、毎回設定しておきます。

```
double kekka_jikan = 0;
char kekka_str[10];

(kekka_jikan の算出処理)

if(kekka_jikan < 40) { strcpy(kekka_str,"優秀！"); }
else if(kekka_jikan < 60) { strcpy(kekka_str,"普通"); }
else { strcpy(kekka_str,"遅い"); }
printf("判定は・・・%sです！", kekka_str);
```

判定結果は今回も適当な値なので、自分で何度か試してみたのちに調整してみてください。

まとめ

特に新しい学習内容はありませんでしたが、配列を一時格納用として利用して、面白いプログラムが書けたと思います。

今回のプログラムのように、for文の初期値や終端値が複雑になる場合は、変数の値をよく確認しながら作成しましょう。

第 **5** 日

山手線ゲームを作ろう

1時限目　プログラム実行時の引数を学ぼう

2時限目　ポインタを理解しよう

3時限目　山手線ゲームを完成させよう

お題に沿った答えを順番に答えていく山手線ゲームを作ります。
答えはデータとしてプログラム内部で持っているので、プレイヤーの答えがあっているかどうかはそのデータに「ポインタ」という概念を使ってアクセスし、判定を行います。
また、プログラム実行時に引数を使う方法もここで学習しましょう。

今日作るプログラムについて

山手線ゲーム

山手線ゲームとは、手拍子にあわせてお題に沿った答えを順番に答えていくゲームです。お題が「色の名前」だったら「赤」「青」「黄」……と色の名前を順にあげていき、答えられなかったり、以前に出た内容と同じものを答えてしまったりした場合は、その人の負けになります。
本日作成する山手線ゲームでは、お題と答えのデータをプログラムで設定しておきます。データの数と順番はきまっていて、コンピュータとプレイヤーが交互に答えます。データのはじまりの位置はランダムにきまります。
プレイヤーが途中で間違った答えを入力した場合はプレイヤーの負け、間違えることなく最後までデータが出おわったら、プレイヤーの勝ちとします。

なお、ゲームのタイトルになっている「山手線の駅名」をお題にすると山手線を利用しない人には不利なため、今回は「星座の名前」をお題にします。

山手線ゲームの実際の動作

1 山手線ゲームプログラムを実行すると、ゲームタイトルとお題のあとに、コンピュータの答えが表示される

```
C:¥source>yamanotesen.exe
古今東西山手線ゲ～～ム！
お題：星座の名前
コンピュータの番　ちゃん　ちゃん！ > おうし
　プレイヤーの番　ちゃん　ちゃん！ >
```

2 答えを入力して［Enter］キーを押す

```
C:¥source>yamanotesen.exe
古今東西山手線ゲ～～ム！
お題：星座の名前
コンピュータの番　ちゃん　ちゃん！ > おうし
　プレイヤーの番　ちゃん　ちゃん！ > ふたご
```

3 コンピュータが答え、またプレイヤーの番となる

```
コンピュータの番　ちゃん　ちゃん！　>　おうし
　プレイヤーの番　ちゃん　ちゃん！　>　ふたご
コンピュータの番　ちゃん　ちゃん！　>　かに
　プレイヤーの番　ちゃん　ちゃん！　>
```

4 コンピュータとプレイヤーで交互に答えあう。ゲーム実行時に「-soto」オプションをつけると答えは逆まわりになる

```
C:¥source>yamanotesen.exe -soto
古今東西山手線ゲ～～ム！
お題：星座の名前
コンピュータの番　ちゃん　ちゃん！　>　てんびん
　プレイヤーの番　ちゃん　ちゃん！　>　おとめ
```

5 全てのデータが出おわると、ゲームはプレイヤーの勝ちとして終了する。答えを間違うとプレイヤーの負けとして終了する

```
(略)
　プレイヤーの番　ちゃん　ちゃん！　>　おひつじ
あなたの勝ち！！
```

```
(略)
コンピュータの番　ちゃん　ちゃん！　>　おひつじ
　プレイヤーの番　ちゃん　ちゃん！　>　おうし
あなたの負け！！
```

第5日 1時限目

【山手線ゲームを作ろう①】

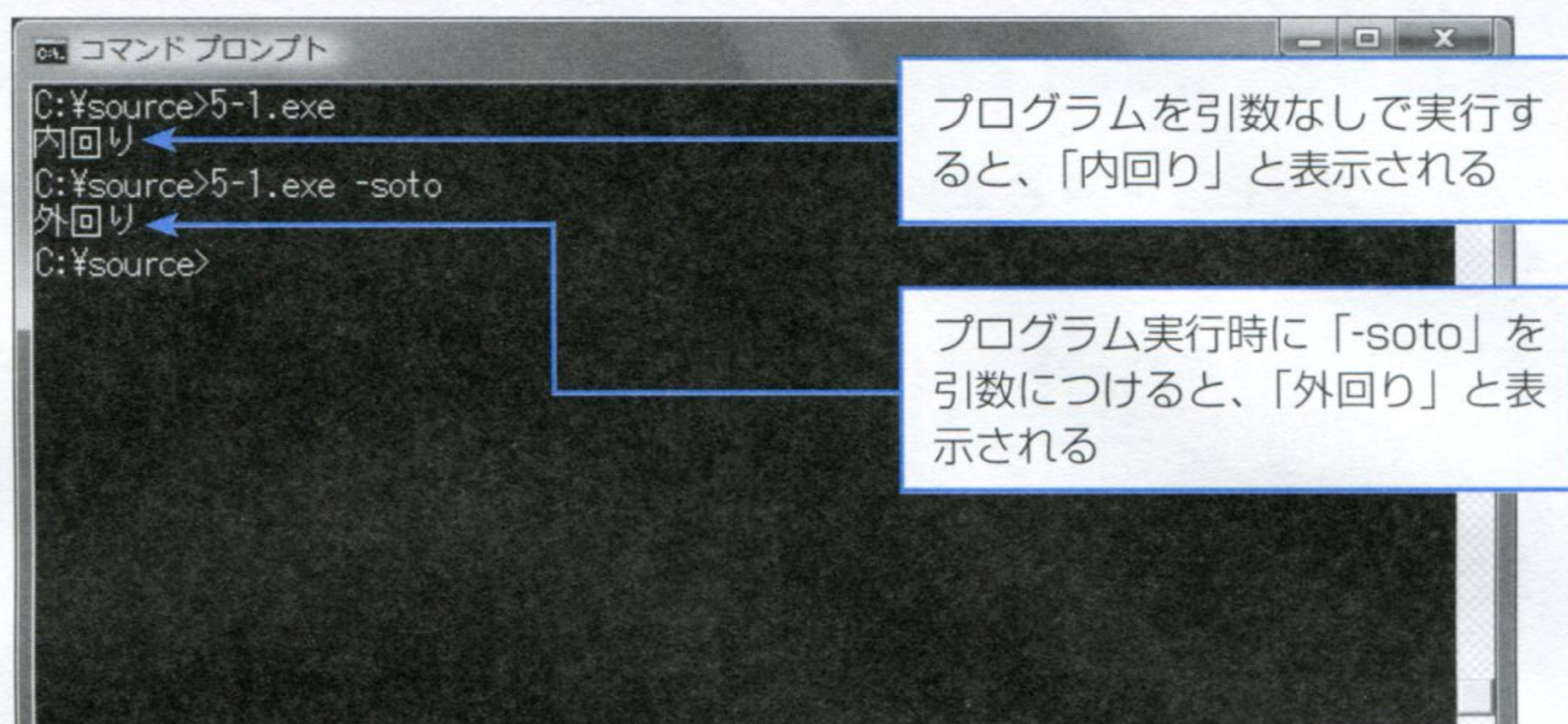
プログラム実行時の引数を学ぼう

プログラムを実行するとき、何か値を引数としてプログラムに渡すことができます。ここではその引数を利用して、「内回り」「外回り」と実行結果をかえてみます。

今回作成する例題

```
C:¥source>5-1.exe
内回り
C:¥source>5-1.exe -soto
外回り
C:¥source>
```

プログラムを引数なしで実行すると、「内回り」と表示される

プログラム実行時に「-soto」を引数につけると、「外回り」と表示される

サンプルファイルはこちら　10days_c ▶ day05-01 ▶ 5-1.c

●このレッスンのねらい

プログラムに値を渡す場合、これまではプログラムの実行中にscanf関数などを使用していました。C言語のプログラムでは、プログラムの実行中だけでなく、プログラムを実行するときに引数として値をプログラムに渡すことができます。本レッスンでは、プログラムの実行時に引数を渡して、プログラム側でそれを受け取る方法について説明します。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int mawari = 1;

    if ((argc > 1) && (strcmp(argv[1], "-soto") == 0)) {
        mawari = 0;
    }

    if(mawari == 1) { printf("内回り"); }
    else { printf("外回り"); }
    return 0;
}
```

2 入力できたら、「5-1.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント
*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、5-1.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 5-1 5-1.c
```

4 プログラムを実行する。プログラム実行時に「-soto」を引数として渡すと「外回り」、それ以外だと「内回り」が表示されれば成功！

```
C:¥source>5-1.exe
```

▼

```
内回り
```

```
C:¥source>5-1.exe -soto
```

```
外回り
```

解説

1 引数とは

C言語で「引数」とは、関数やプログラムに渡すデータのことを指します。

(1) 関数の引数

関数の引数については、これまでにも何度か出てきました。関数名のあとの( )の中に書いたものが引数で、その中のデータを関数に渡しています。

```
printf("Hello!");
        ↑
       引数
```

この場合は、文字列「Hello!」が引数です。printf関数に渡すと、この文字列を出力してくれます。引数が複数ある場合は、次のように引数をカンマ「,」で区切ります。

```
printf("%d %s", 128, "Hello!");
          ↑       ↑       ↑
       第1引数  第2引数  第3引数
```

最初の引数「%d %s」を第1引数、次の128を第2引数といいます。以降、第3引数、第4引数…、と続きます。

(2) コマンド引数

関数の他に、コマンドプロンプト上でも、コマンドに何かデータを引数として渡すことができます。その場合は、コマンドに続いてひとつ以上のスペースを空けて書きます。例えば、ディレクトリ移動を行うcdコマンドでも、引数を使います。

```
C:¥>cd ¥source
```

cdコマンドのあとにひとつ以上のスペースを空けて、ディレクトリ名を引数として書きます。これで引数に指定した「¥source*2」ディレクトリに移動します。

ヒント
*2:「¥」はディレクトリの区切りを表します（第1日3時限目参照）。

プログラムにも引数を渡すことができます。コンパイルを行うプログラムであるgccに渡す引数は、「コンパイルを行いたいファイル名」です。

```
C:¥source>gcc sample.c
```

ファイル名「sample.c」がプログラムgccへの引数です[*3]。コマンドプロンプト上での引数は、コマンドやプログラム名のあとに、ひとつ以上のスペースを空けて書きます。複数あるときは、さらにひとつ以上のスペースを空けて続けて書きます。

```
C:¥>copy file1.txt file2.txt
```

コマンド　第１引数　第２引数

ヒント
*3：gcc実行時のオプションである-oや-Wallも、ひとつの引数です。

2 C言語のプログラムに引数を渡す

自分で書いたプログラムにも、引数を渡すことができます。ここでは、渡された引数をプログラム側で受け取る方法についても解説します。

(1) 引数を渡す

C言語で書いたプログラムにも、引数を渡すことができます。プログラムを実行するときに、プログラム名に続いて

```
C:¥source>sample.exe 1 2 3
```

プログラム　引数

と書けば、数値「1」「2」「3」が複数の引数として、sample.exeにそれぞれ渡されます。

(2) 引数を受け取る

今度はプログラム側で、その引数を受け取る方法を見てみます。受け取る方法は、main関数のあとの()の中に、

【プログラム側で引数を受け取る】

```
int main(int argc, char* argv[]) {
```

と書いておくだけです[*4]。int型変数argcには引数の個数が入ってきます。正確にはプログラム名を含めた個数なので、

```
C:¥source>sample.exe abc de f
```

プログラム名　引数　引数　引数

ヒント
*4：このように書いておくと、main関数の中でint型変数argcと、配列argvを使うことができます。この部分は、第７日の関数の項目で学習します。

と実行すると、変数argcには4という数字が入ります。main関数の中では、int型変数argcの値は4、と参照できます。

引数の中身は、配列argvに入っています。しかし、よく見るとただの配列ではなさそうです。char型宣言のあとにある「*」は、ポインタを意味します。ポインタについては2時限目で詳しく説明するので、今はあまり深く考えないで大丈夫です。

プログラム実行時に渡した第1引数は、argv[1]として参照することができます。つまりこの場合、プログラム名を含めた引数は、次のように配列argvに格納されます。

```
C:¥source>sample.exe abc de f
```

argv[0]　argv[1]　argv[2]　argv[3]

ここで、文字列について復習してみましょう。文字列を扱うには、文字型配列を利用します。文字型配列では、それぞれの要素には1文字しか入っていません。

```
char str[] = "hoge";
```

この場合、str[1]には文字「o」、str[2]には文字「g」が入っています。ですが、

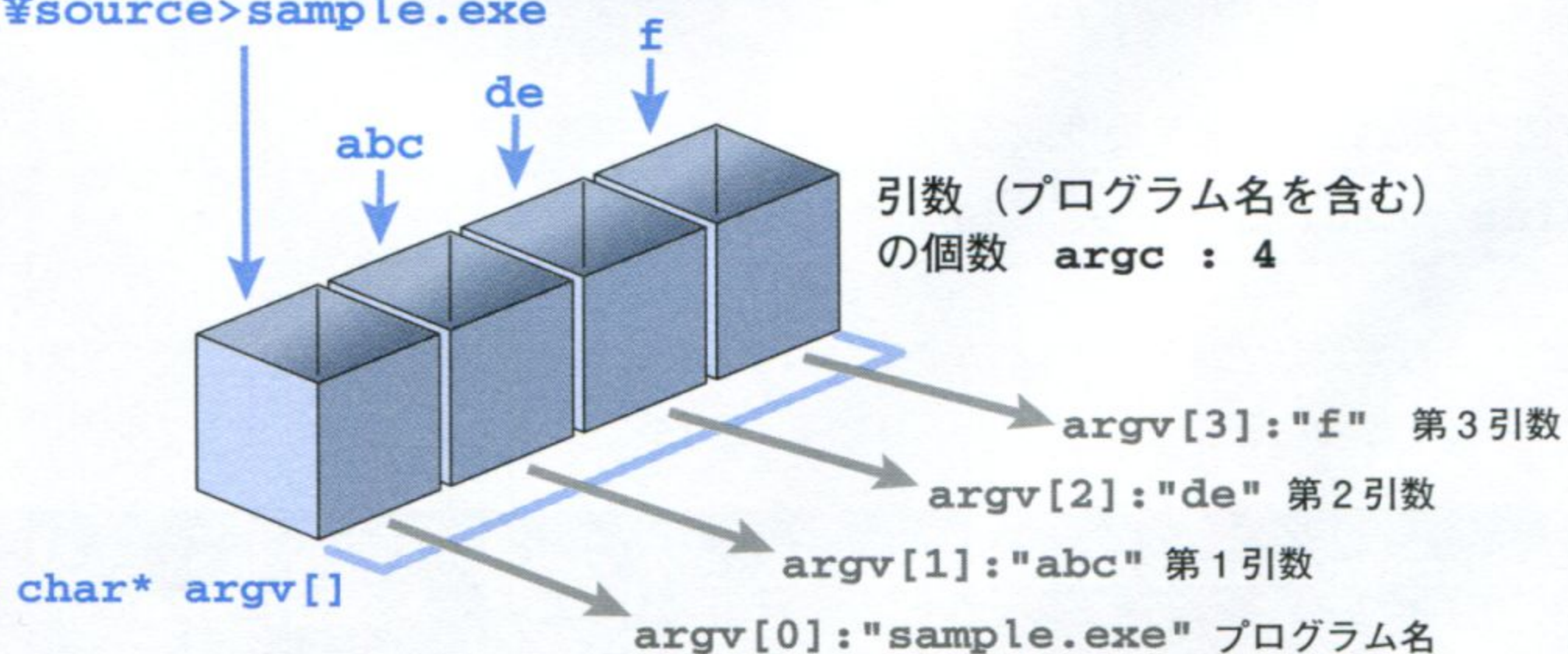
```
char* argv[]
```

と書くと、argv[i]（iは添え字番号）で、1文字ではなく文字列を参照できます。これが、ポインタというものを使った魔法です。

ポインタについての原理の詳細は、2時限目に詳しく説明します。

●引数の受け取り

引数受け取り：
main (int argc, char* argv[])

C:¥source>sample.exe
abc
de
f
引数（プログラム名を含む）
の個数　argc : 4
argv[3]:"f"　第3引数
argv[2]:"de"　第2引数
argv[1]:"abc"　第1引数
argv[0]:"sample.exe"　プログラム名
char* argv[]

(3) サンプルプログラムを使って確認する

引数を読み込んで表示するプログラムを書きましょう。

【5-1_sample1.c】

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;

    printf("プログラム名：%s¥n", argv[0]);
    for(i = 1; i < argc; i++) {
        printf("第%d引数 : %s¥n", i, argv[i]);
    }
    return 0;
}
```

この5-1_sample1.cをコンパイルし、引数を渡して実行します。

```
C:¥source>5-1_sample1.exe aaa 2 4
```

```
プログラム名：5-1_sample1.exe*5
第１引数 : aaa
第２引数 : 2
第３引数 : 4
```

ヒント
*5：コンパイラの種類によっては、プログラム名は「C:¥source¥5-1_sample1.exe」と絶対パスで表示される場合があります。

変数argcの値は4なので、argv[1]～argv[3]に引数が格納されています。引数の出力は文字列なので、ここでは書式指定を「%s」にしています。

ひとつ、ここで注意しなければならないことがあります。引数は、

```
char* argv[]
```

と文字型*6を指定しています。したがって、例え引数に数値を渡しても、その引数をプログラム側で受け取ると文字列扱いになります。引数に数値5を入力してもプログラム側では文字の「5」となってしまう、ということです。もし数値に直したいときは、atoi関数を使いましょう。

ヒント
*6：これは、正確にはポインタを使った文字型です。

3 引数の書き方

プログラムの引数が複数ある場合、ひとつ以上のスペースで区切ります。つまり、スペースがあると引数の区切りとみなされてしまいます。

スペースを含んだデータを引数に指定したい場合は、ひとつの引数をダブルクォート「"」で括ります。

```
C:¥source>5-1_sample1.exe "Hello World!"
```

これで、文字列「Hello World!」が第1引数に指定されます。シングルクォート「'」で括ると別のデータとみなされるので、注意してください。

```
C:¥source>5-1_sample1.exe 'Hello World!'
```

この場合、第1引数に「'Hello」が、第2引数に「World!'」が指定されたことになります。

ダブルクォートをデータに含めたい場合は、引数を括るダブルクォート以外、ダブルクォートの前に「¥」をつけます。

```
C:¥source>5-1_sample1.exe "Hello ¥"World!¥""
                           ↑
                        引数の中身
```

最初と最後のダブルクォートは、引数を括る意味で使われています。中の2つはデータとしてのダブルクォートなので、前に「¥」をつけています。よって、この場合の第1引数は、「Hello "World!"」になります。

4 外回りの指示を受け取るプログラム

本日作成する山手線ゲームは、登録してあるデータをコンピュータとプレイヤーで交互に答えていくゲームです。開始場所はランダムですが、そこから配列のうしろ方向に進むか前方向に進むかを指定できるようにします。

【山手線データの場合*7】

```
char *data[] = { "東京", "神田", "秋葉原", "御徒町", "上野",
          ←外回り （スタート） 内回り→
```

【星座データの場合】

```
char *data[] = { "おひつじ", "おうし", "ふたご", "かに", "しし",
          ←外回り （スタート） 内回り→
```

内回り（配列うしろ方向順）を基本にし、プログラムの実行時に「-soto」という引数が渡された場合のみ、外回り（配列前方向順）にします。

引数を指定し忘れていた場合は、argv[]にはプログラム名しか入っていません。引数が入っているはずのargv[1]には何も入っていないので、参照しようとすると環境によってはエラーになります*8。

ヒント

*7：他のデータを使用する場合も同様です。

ヒント

*8：配列は、値の定義されていない要素を参照しようとすると、環境によってはエラーになります。

よって、argcで引数の有無とその個数を先に確認してから、処理を分岐しておく必要があります。argcの値が1のときは、プログラム名だけです。argcが1よりも大きければ、何か引数があるということになります。

```
C:¥source>5-1.exe          ←argc は 1
C:¥source>5-1.exe a b      ←argc は 3
```

引数がある場合に、その値を「-soto」と比較します。

```
int mawari = 1;

if ((argc > 1) && (strcmp(argv[1], "-soto") == 0)) {
    mawari = 0;
}
```

論理演算子&&を使うと、(argc > 1)が偽だった場合は、次の(strcmp(argv[1], "-soto") == 0)の判定は行いません。
両方が真の場合は、データ参照順序を外回り（配列前方向順）に変更します

まとめ

プログラム実行時の引数について学習しました。
プログラムの中で引数を受け取らない場合は、今までどおりmain関数のあとの( )には何も書く必要はありません。必要なときだけ、変数argc、argvを使います。

第5日 2時限目

ポインタを理解しよう

【山手線ゲームを作ろう②】

1時限目で学習した引数の受け取りには、ポインタを使いました。「C言語最大の難関」と思われがちなのが、この「ポインタ」です。

今回作成する例題

```
C:¥source>5-2.exe
スタート位置 7 0022FF3C
さそり
いて
やぎ
みずがめ
うお
おひつじ
おうし
ふたご
かに
しし
おとめ
てんびん

C:¥source>
```

配列のスタート位置と、そこから順にすべての要素を表示する

サンプルファイルはこちら 10days_c ▶ day05-02 ▶ 5-2.c

●このレッスンのねらい

難しいといわれがちなポインタですが、実際はそれほど難しいものではありません。そして、好んで頻繁に使うものでもないと思います。無理に使う必要もありません。
文字列の配列を作る場合にはよく利用しますが、ポインタということはあまり意識せずに使うことができます。プログラム実行時の引数の扱いがその例です。
しかし、原理を理解しているほうがよいことにかわりはありません。この時限では、ポインタのしくみについてじっくり学習しましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    char *data[] = { "おひつじ", "おうし", "ふたご", "かに",
                     "しし", "おとめ", "てんびん", "さそり",
                     "いて", "やぎ", "みずがめ", "うお" };
    int data_len = 12;
    int i, p, start;

    srand(time(NULL));
    start = rand()%data_len;

    printf("スタート位置 %d %p¥n", start, data+start);

    for(p = start, i = 1; i <= data_len; i++) {
        printf("%s¥n", *(data+p));
        p++;
        if(p == data_len) { p = 0; }
    }
    return 0;
}
```

2 入力できたら、「5-2.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント
*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、5-2.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 5-2 5-2.c
```

4 プログラムを実行する。配列のスタート位置と、そこから順にすべての要素が表示されれば成功！

```
C:¥source>5-2.exe
```

```
スタート位置 7 0022FF3C
さそり
いて
やぎ
みずがめ
うお
おひつじ
おうし
ふたご
かに
しし
おとめ
てんびん
```

解説

1 アドレスのしくみ

ポインタの説明の前に、アドレスについて説明します。

(1) アドレスとは

アドレスとは、「番地」を意味します。

C言語のプログラムの中で、ある変数を使う場合について考えます。

```
int d = 5;
```

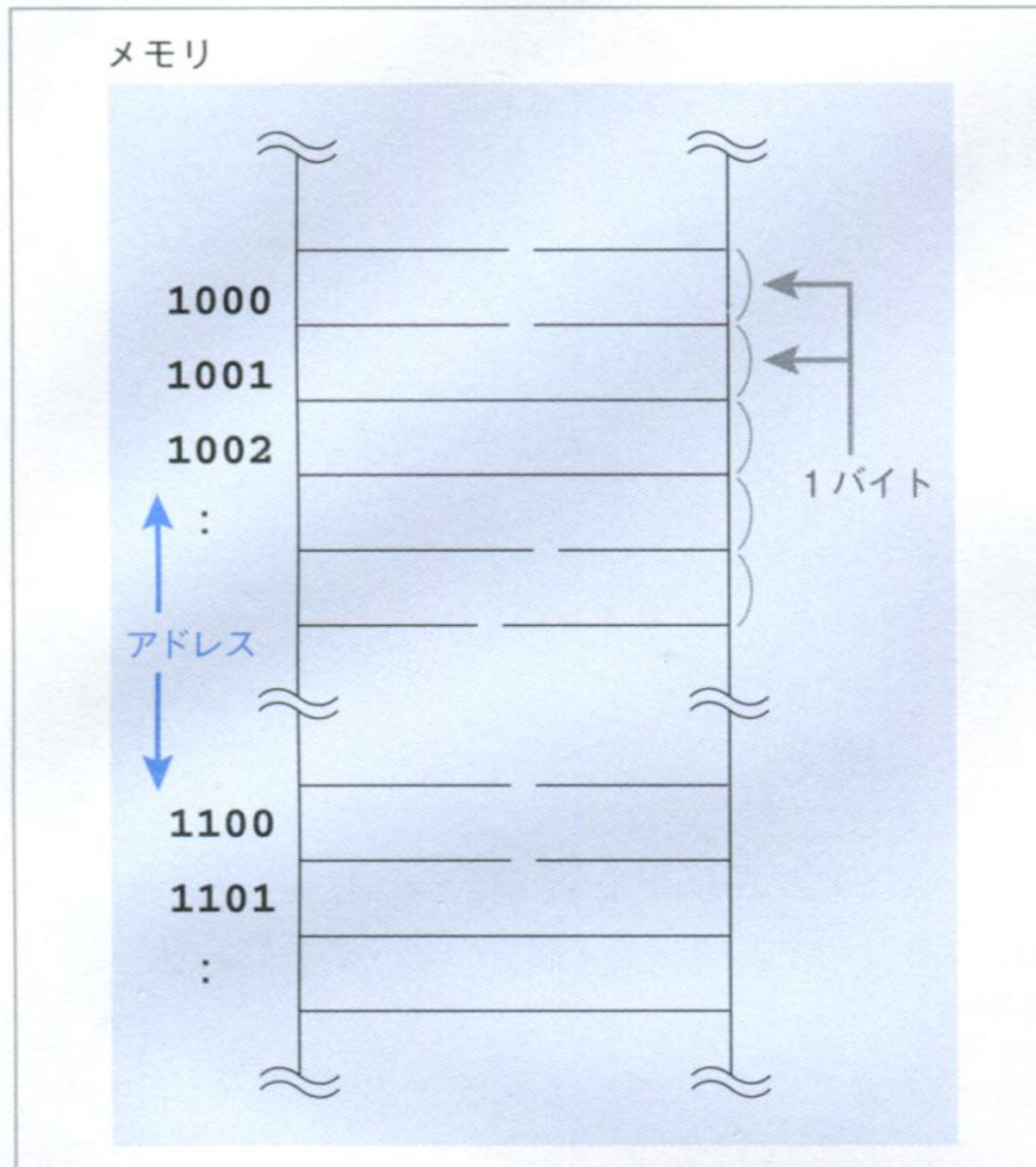
この場合、整数型の変数dの値は5です。この宣言のあと、プログラムの中で変数dを参照すると、ずっと「5」という数字が取得できます。また、ある時点で変数dに「6」を代入すれば、今度は「6」が変数dの値になります。

プログラムを実行すると、変数dは実行したコンピュータのメモリ上の適当な場所に割り振られ、そこに値が記憶されます。この割り振られた場所の位置を表すのがアドレスです。このあと、同じプログラム中で変数dを参照すると、このアドレスからdの値を取得できます。変数dに6を代入しても、アドレスにあるdの値が変更されるだけで、変数dの値が格納されているアドレスはかわりません。

(2) メモリの割り振り

コンピュータ上で一時的にデータを記憶しておく領域を、メモリと呼びます。メモリは1バイトずつに区切られていて、その区切られた各場所に、順に番号をつけています。この番号がアドレスです。アドレスは番号なので、何桁かの数値で表されていています。桁数は実行するコンピュータにより異なります。

●メモリ内のアドレスの割り振り

メモリ
1000
1001
1002
:
アドレス
1100
1101
:
1バイト

プログラムは、実行時にメモリ上の各場所（アドレス）にそのプログラムで使う変数や関数を割り振ります。そこにデータを保存しておくことにより、プログラム実行中に変数の値をずっと保持しておくことができるのです。アドレスを利用して、プログラムは効率よく実行されるしくみになっています。

例えば、次の3つの変数を利用するプログラムを実行すると、変数d1、d2、cが、メモリ上にそれぞれ割り振られます。

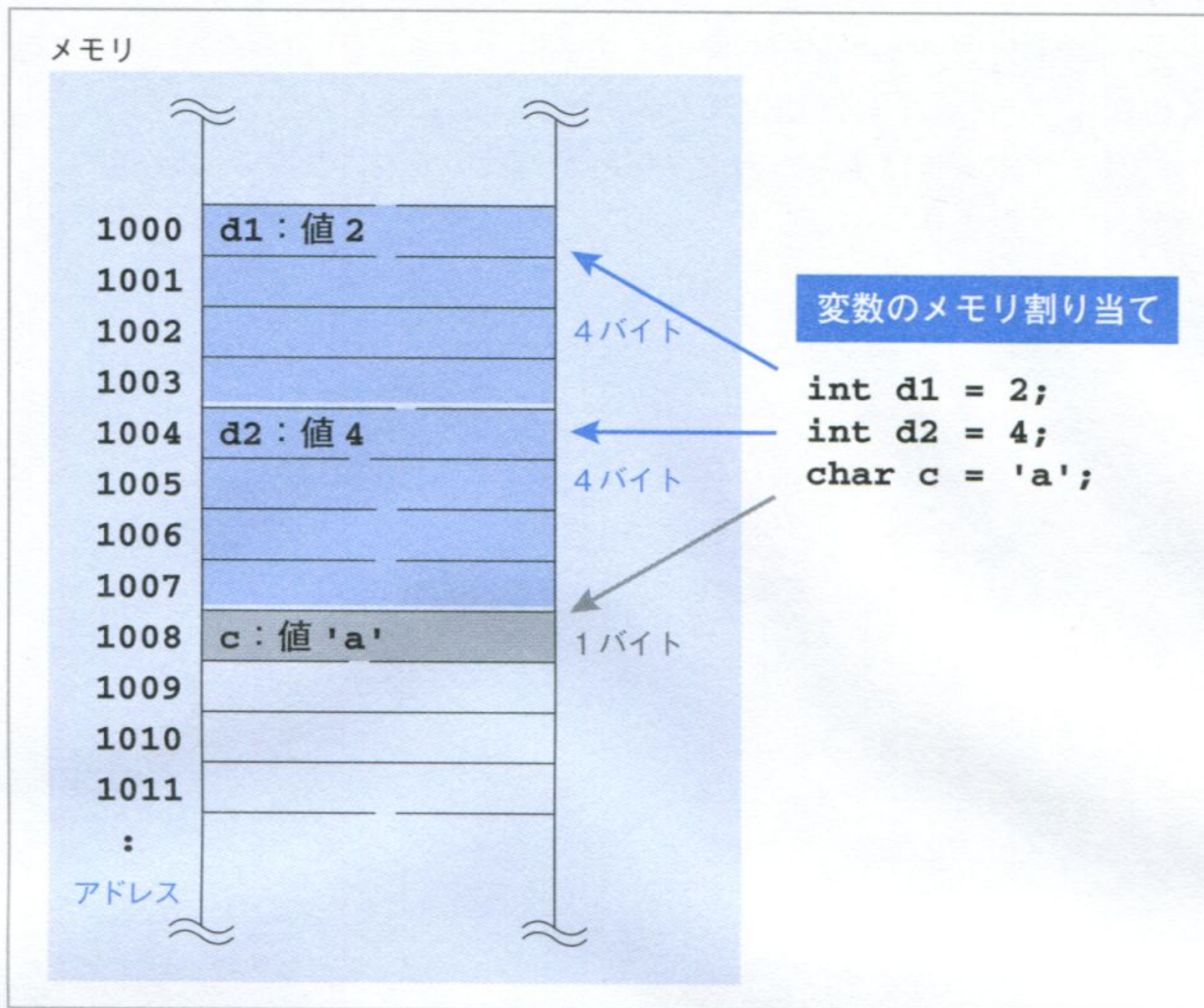
```
int d1 = 2;
int d2 = 4;
char c = 'a';
```

●変数のメモリ割り当て

メモリ

アドレス		
1000	d1：値 2	4バイト
1001		
1002		
1003		
1004	d2：値 4	4バイト
1005		
1006		
1007		
1008	c：値 'a'	1バイト
1009		
1010		
1011		
：		

変数のメモリ割り当て

```
int d1 = 2;
int d2 = 4;
char c = 'a';
```

割り振られた位置（アドレス）は、実行環境により異なります。変数d1はint型変数です。int型のデータは4バイトです。つまり、メモリの1区間をを4個分（4バイト分）使って値を保存します。

上の図では、変数d1のアドレスが1000番地[*2]になりますが、データの実体は1000～1003の区間を使って保存されています。変数d2も同様です。int型なので1004～1007番地を使っていますが、アドレスは1004番地を指します。つまり、データが保存されている領域の先頭アドレスが、その変数のアドレスということになります。

最後の変数cはchar型なので1バイト、つまり、メモリの1区間だけを使って保存されます。アドレスは1008です。

プログラムのデータを保存するには、データの型の大きさ分、メモリを必要とすることをおぼえておいてください[*3]。

ヒント

*2：実際にアドレスの値を参照すると、16進数で表現されます。今のところはわかりやすいように、10進数の数値を使って説明します。

ヒント

*3：コンピュータ上で現在動いているソフトのデータは、すべてメモリ上に同じように割り振られています。

COLUMN

C言語の移植性について

本書の冒頭で、C言語の特徴として「プログラムのソースを他のOSに移植することが容易である」と書きましたが、これは「ソース」のみです。あるコンピュータ上で作った実行ファイルを、別のOSのコンピュータ上に持っていっても、動きません。これは、ソフトの実行形式やアドレスの割り振り方などがコンピュータやコンパイラにより異なるからです。

ただし、ソースを別のOSでコンパイルし直すと、動きます。これが「移植が容易」という意味です。逆にいうと、同じOS、同じコンピュータの種類でコンパイルしたものは、別のコンピュータ上でもコンパイルし直すことなく、実行ファイルを動かすことができます。

(3) 実際にアドレスを参照してみる

アドレスは、変数名の前に「&」をつけると参照できます。試してみましょう。

【5-2_sample1.c】

```
#include <stdio.h>

int main() {
    int d = 5;
    int a = 10;
    char c = 'a';

    printf( "dのアドレス：%p 値：%d¥n", &d, d );
    printf( "aのアドレス：%p 値：%d¥n", &a, a );
    printf( "cのアドレス：%p 値：%c¥n", &c, c );
    d = 6;
    printf( "dのアドレス：%p 値：%d¥n", &d, d );
    return 0;
}
```

▼

```
dのアドレス：0022FF8C 値：5
aのアドレス：0022FF88 値：10
cのアドレス：0022FF87 値：a
dのアドレス：0022FF8C 値：6
```

アドレス表示の書式には%pを使い、表示は16進数になります。

このプログラムで、各変数が格納されている位置（アドレス）を見ることができます*4。途中で変数dの値が5から6に変わりましたが、アドレスはそのままです。

ヒント

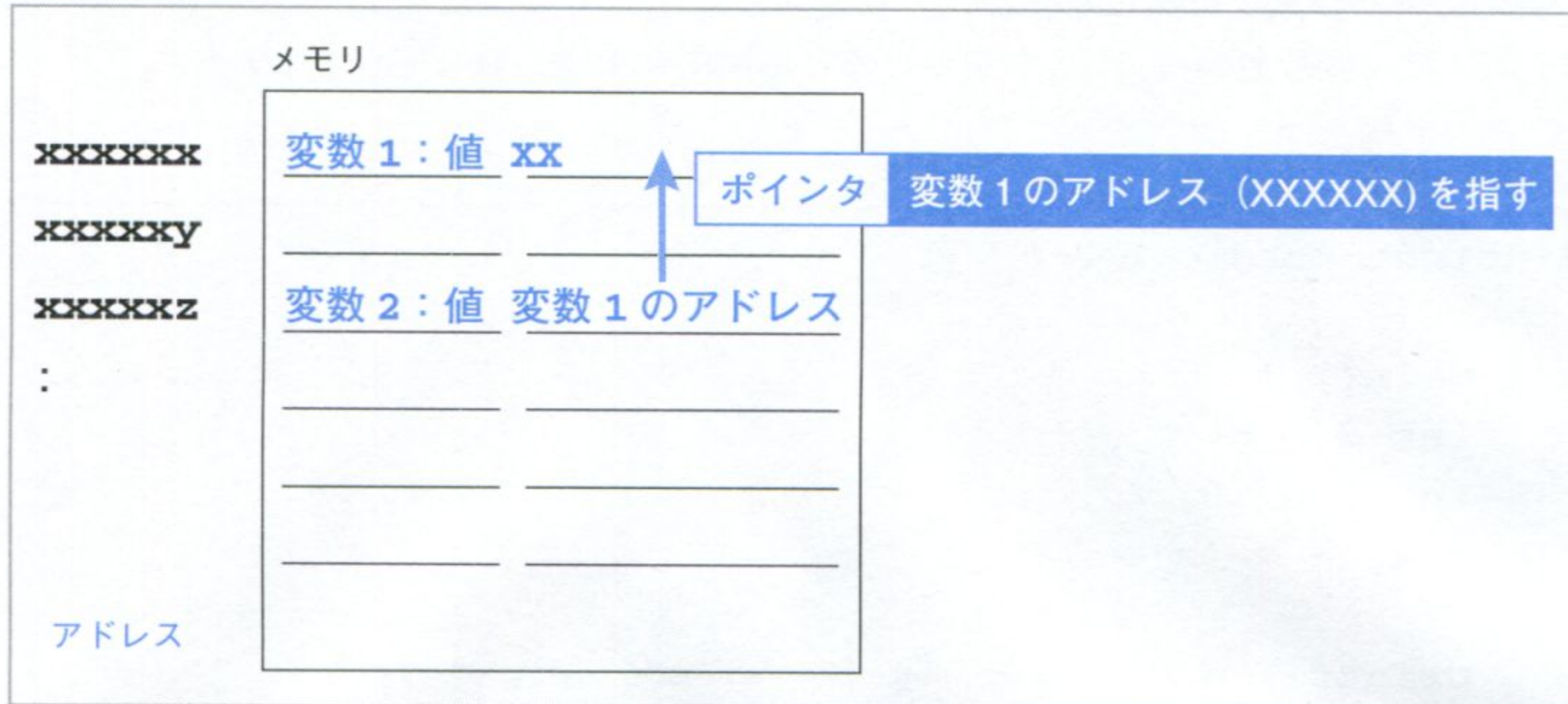
*4：アドレスの値は、実行する環境によって異なります。

2 ポインタのしくみ

アドレスを理解したうえで、いよいよポインタについて説明します。

(1) ポインタの基本

さて、アドレスの存在だけ知っていても、あまり意味はありません。通常、メモリに割り振られた変数の場所には値が格納されていますが、値のかわりに「別の変数が格納されているアドレス」を保存しておくこともできます。値のかわりにアドレスを保存しておくと、そのアドレスを指し示していることになります。これがポインタです。

●ポインタの正体

ある変数のアドレスを値として持つ変数を、「ポインタ変数」といいます。ポインタ変数の宣言は、変数名の前に「*」をつけて書きます。

【ポインタ変数の宣言①】

```
int *p;
```

または、

【ポインタ変数の宣言②】

```
int* p;
```

と書きます[*5]。これで、int型のデータが格納されているアドレスを指すポインタ変数ができました。この変数に、別のint型変数dのアドレスを代入します。

変数のアドレスは、変数名の前に「&」をつけて表します。

ヒント

5：「」を変数名のすぐ前につけるか、型のうしろにつけて書くかは、個人の自由です。

```
int *p;
int d = 5;
p = &d;        // 変数dのアドレスをポインタ変数pに格納する
```

こうすると、そのあとのプログラムでは、

```
p     変数dのアドレス
*p    変数dの値（指し示すアドレスの値）
```

として、それぞれの値を参照できます。簡単なプログラムで確認してみましょう。

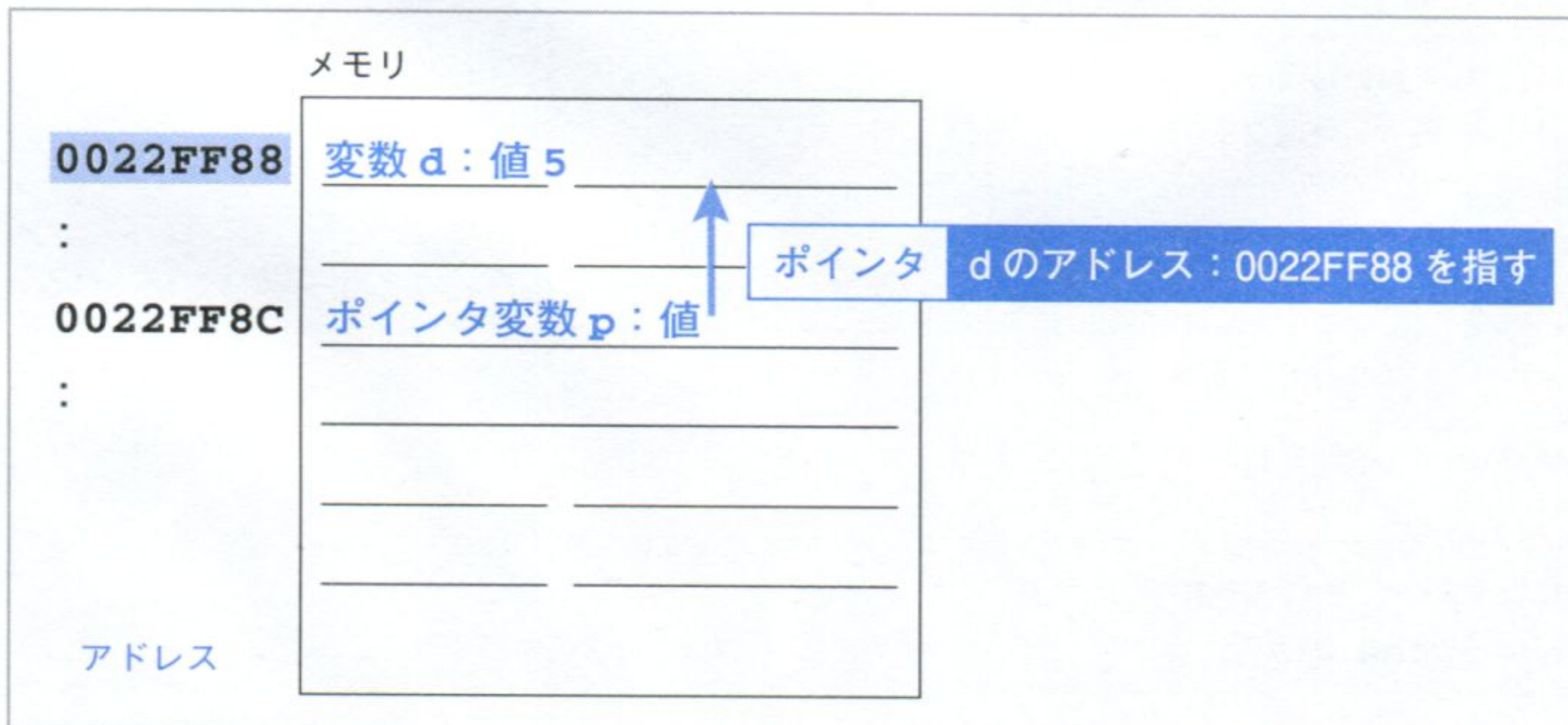
【5-2_sample2.c】

```
#include <stdio.h>

int main() {
     int *p; // ポインタ変数
     int d = 5;

     p = &d; // 変数dのアドレスをポインタ変数pに格納する
     printf( "dのアドレス：%p 値：%d¥n", &d, d );
     printf( "pのアドレス：%p¥n", &p );
     printf( "pが指すアドレス：%p pの値：%d¥n", p , *p);
     d = 6;
     printf( "pが指すアドレス：%p pの値：%d¥n", p , *p);
     return 0;
}
```

```
dのアドレス：0022FF88 値：5
pのアドレス：0022FF8C
pが指すアドレス：0022FF88 pの値：5
pが指すアドレス：0022FF88 pの値：6
```

ポインタ変数pが指しているのは変数dのアドレスで、*pの値は変数dの値になっています。

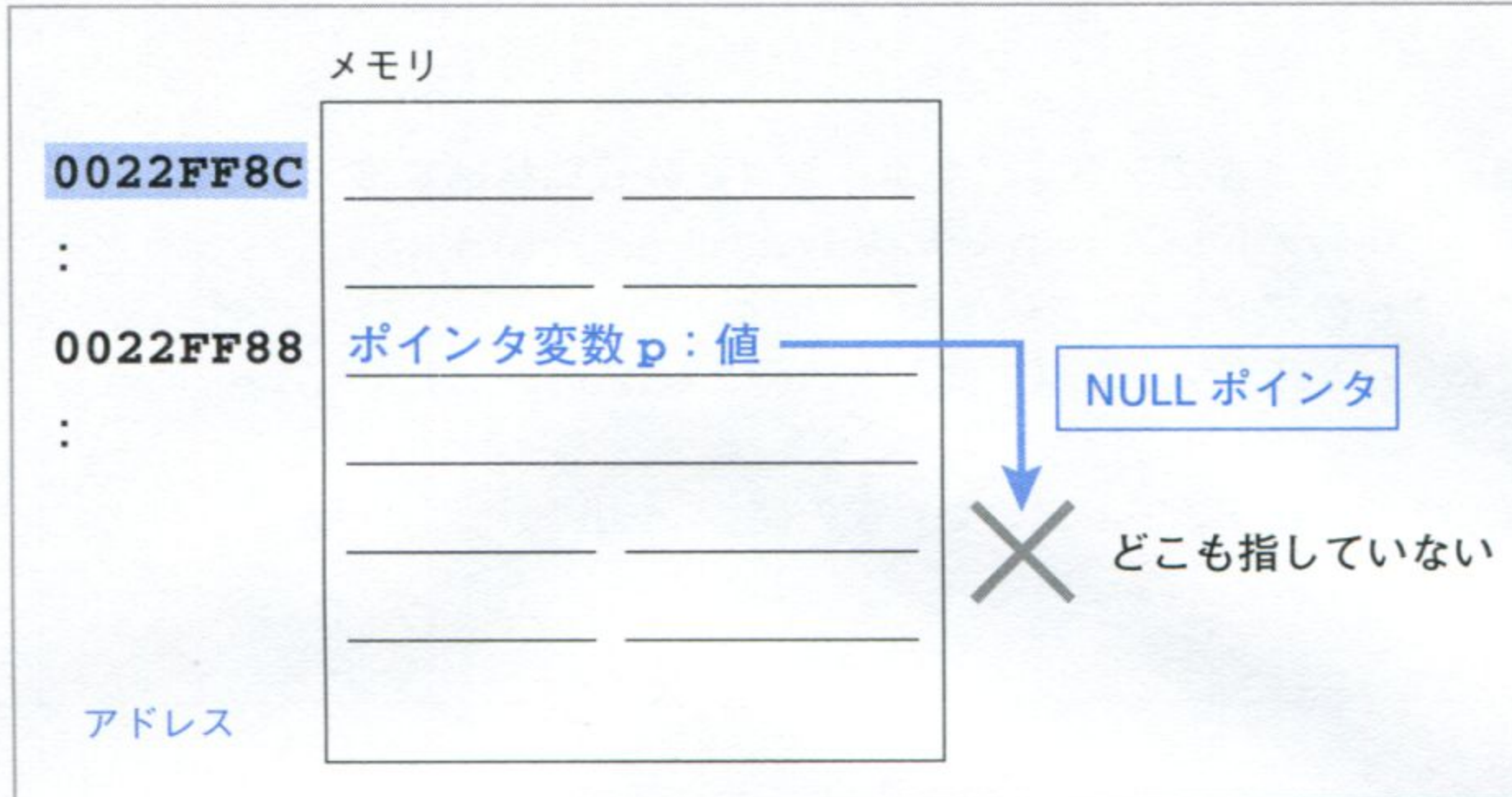
●ポインタと値の参照

メモリ
0022FF88 変数d：値5
:
0022FF8C ポインタ変数p：値
:
ポインタ dのアドレス：0022FF88を指す
アドレス

(2) NULLポインタ

ポインタ変数を初期化するときに、「(現在) どこも指していない」という意味の、NULLを代入しておくこともできます。

【ポインタ変数をNULLで初期化】

```
int *p = NULL;
```

●ポインタ変数にNULLを代入

(3) ポインタとデータ型

先ほどはint型の変数を指し示すポインタ変数を作りました。int型に限らず、他のデータ型でも、ポインタ変数を作り出せます。ただし、ある変数のデータ型と、それを指し示すポインタ変数のデータ型は、同じである必要があります。

```
int *p; // ポインタ変数
double d = 5.2;
p = &d;
```

×

変数dとそれを指し示すポインタ変数pのデータ型が一致していない

```
double *p; // ポインタ変数
double d = 5.2;
p = &d;
```

○

変数dとそれを指し示すポインタ変数pのデータ型が一致している

文字列のポインタを作ってみましょう。

```
char *str = "test";
str = "change";
```

最初に文字列"test"がメモリ上に格納され、ポインタ変数strはその先頭アドレスを指しています。2行目の代入で、今度はポインタ変数strは文字列"change"が格納された先頭アドレスを指すように変更されたことになります。

ポインタ変数に格納するのはアドレスなので、文字列をコピーするstrcpy関数ではなく、イコール「=」を使うことができます。

3 配列とポインタ①

アドレスとポインタについて理解はできたけれど、これがいったい何の役に……？ と思う方もいるでしょう。ここで、ポインタの利点について考えてみましょう。

例えば、複数の人で本を回覧する場合を考えます。

今現在、だれが本を持っているのか知りたい場合は、端から聞いて回るか、最初に読んだ人から順に、次に誰に渡したかを確認していく必要があります。

しかし、誰か一人が「今、誰が本を持っているか」を把握していれば、その人に聞けばすぐにわかります。このケースでいう、その一人だけ把握している人が、「ポインタ」です。「ポインタのおかげ」で、情報をすばやく引き出すことができるのです。

別の例で考えましょう。あなたは、あまりよく知らない場所へ行かなければならなくなりました。そいうときは、地図を頼りに目的の住所を探したり、電柱に書いてある住所を頼りに探したりするでしょう。しかし、目的の住所の近くに知っている場所や建物があれば、「そこから5軒目」とか「その次の道を右に曲がって2軒目」など、特定の場所を基点にして探すことができます。

ポインタの場合も同様です。メモリ上のある基点アドレスをもとに、そこから「ひとつ先」「3つ先」と指定すれば、データを素早く引き出すことができるのです。

(1) 文字型配列

第4日に学習した文字列を思い出してください。文字列は特殊な文字型配列です。

```
char str[] = "hoge";
```

実は、配列の各要素のアドレスは、住所のように続いているのです。

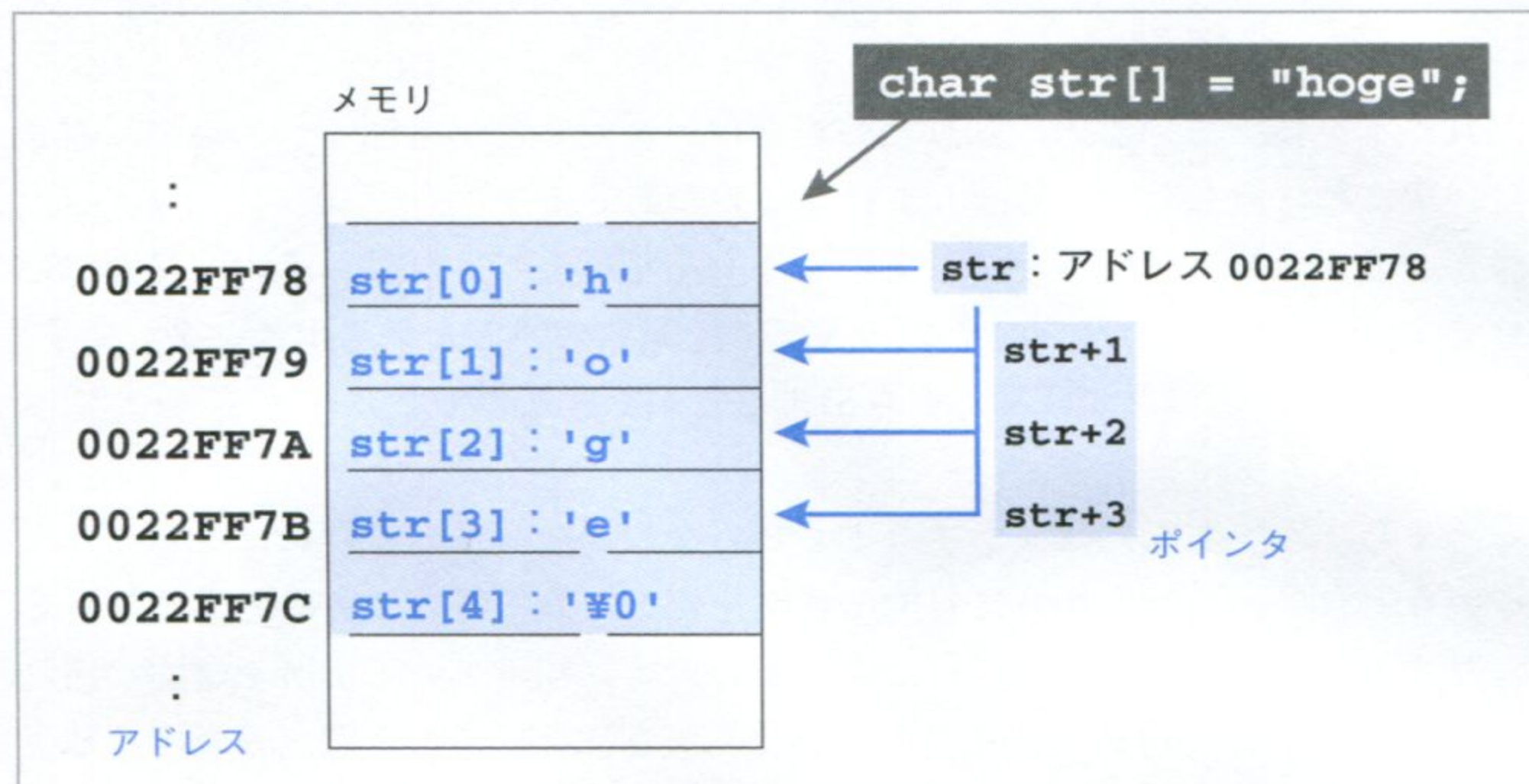
str[2]なら、str[0]を基点にして先に2つ進んだ場所にデータが存在します。str[3]は、str[0]を基点にして3つ先の場所にデータが存在します。「配列str」は、str[0]のアドレスを指しています。配列名は、ポインタの役割をしているのです。

ポインタをひとつずらす（str + 1）と、次のstr[1]のアドレスを指します。よって、それぞれの要素のアドレスは、

```
str       str[0]のアドレス
str + 1   str[1]のアドレス
str + 2   str[2]のアドレス
str + 3   str[3]のアドレス
```

と、表現できます。各要素の値を参照するには、アドレスの中身を示す「*」をつけます。

*(str)	str[0] の値
*(str + 1)	str[1] の値
*(str + 2)	str[2] の値
*(str + 3)	str[3] の値

●文字型配列とポインタ

プログラムで確認してみましょう。

【5-2_sample3.c】

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "hoge";
    int i, s;

    s = strlen(str);
    for(i = 0; i < s; i++) {
        printf( "str[%d]: %c アドレス：%p¥n", i, *(str+i), str+i );
    }
    return 0;
}
```

```
str[0]: h アドレス：0022FF78
str[1]: o アドレス：0022FF79
str[2]: g アドレス：0022FF7A
str[3]: e アドレス：0022FF7B*6
```

ヒント

*6：文字列最後の'¥0'については出力を省略します。

配列データのアドレスが続いていることを確認できます。文字型の配列なので、メモリ1区間に1文字ずつ格納されます。

配列の名前自体がポインタの役割をしているので、配列名の前に「&」をつけず、str+iだけでアドレスを表します。

(2) その他の型の配列

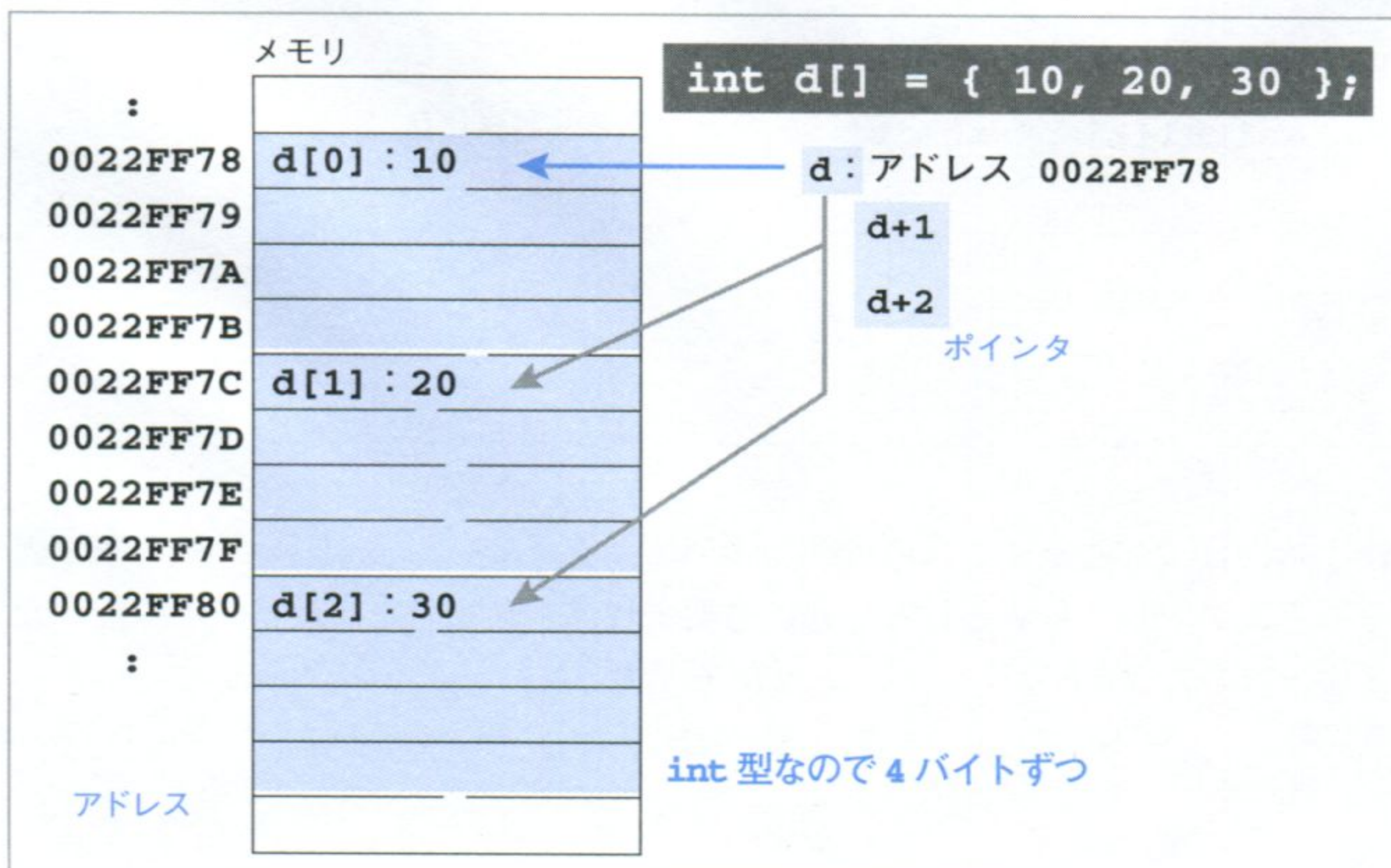
もうひとつ実験してみます。int型配列のデータとアドレスを参照してみます。

```
int d[] = { 10, 20, 30 };
```

```
d[0]: 10 アドレス：0022FF78
d[1]: 20 アドレス：0022FF7C
d[2]: 30 アドレス：0022FF80
```

今度はアドレスが連続した番号ではありません。これは、1アドレスは1バイト分の大きさを表しているからです。文字型は1バイトでなので、文字型配列の場合、ポインタをひとつずらすと1バイト分、つまりアドレスもひとつずつ移動しました。

しかし、整数型は4バイト必要なので、ポインタがひとつずれるとアドレスは4つ先を示します。配列のデータ型によって、アドレスのずれ方が異なります。

●整数型配列のポインタ

4 配列とポインタ②

ポインタは、複数の文字列データを扱いたいときに活躍します。

C言語ではひとつの変数にはひとつの値が入り、複数のデータをまとめて格納するには配列を利用します。

```
int d1 = 5;
int d2[] = { 5, 15, 23 };
```

今度は文字列の場合を考えてみましょう。動物名を格納する変数を作ります。

```
char animal[] = "dog";
```

別のデータが必要となったときは、新しい変数を作ります。

```
char animal2[] = "cat";
```

また別のデータが必要になったら……と延々に繰り返した場合、変数animal、animal2、animal3、animal4……と次々に新しい変数ができてしまいます。

「これを配列にできたら便利なのに！」と誰もが思うことでしょう。例えば、animal[2]を参照したら、文字列「rabbit」が取得できるという具合です。しかし、文字列とは文字型の配列ですから、各要素のアドレスには1文字しか入りません。

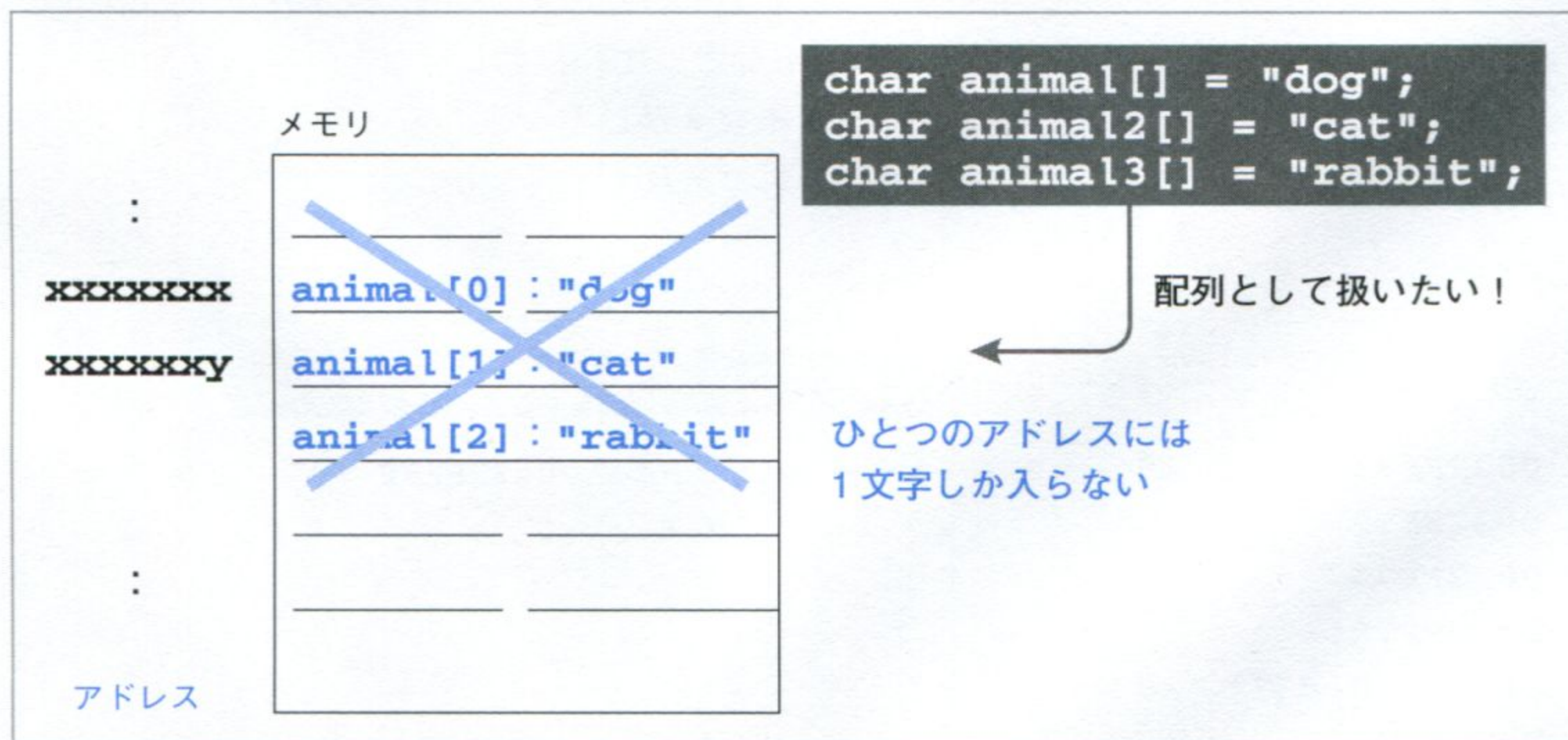
●文字列とアドレスの解決できない問題

```
char animal[] = "dog";
char animal2[] = "cat";
char animal3[] = "rabbit";
```

メモリ

アドレス	メモリ
：	
xxxxxxx	animal[0]："dog"
xxxxxxy	animal[1]："cat"
	animal[2]："rabbit"
：	

配列として扱いたい！

ひとつのアドレスには
1文字しか入らない

これを解決するには、どこかにそれぞれの文字列データ「dog」「cat」「rabbit」を定義しておき、その先頭アドレスを値として配列に格納すれば、動物名を複数格納する配列ができあがります。このような配列が、ポインタ配列です。

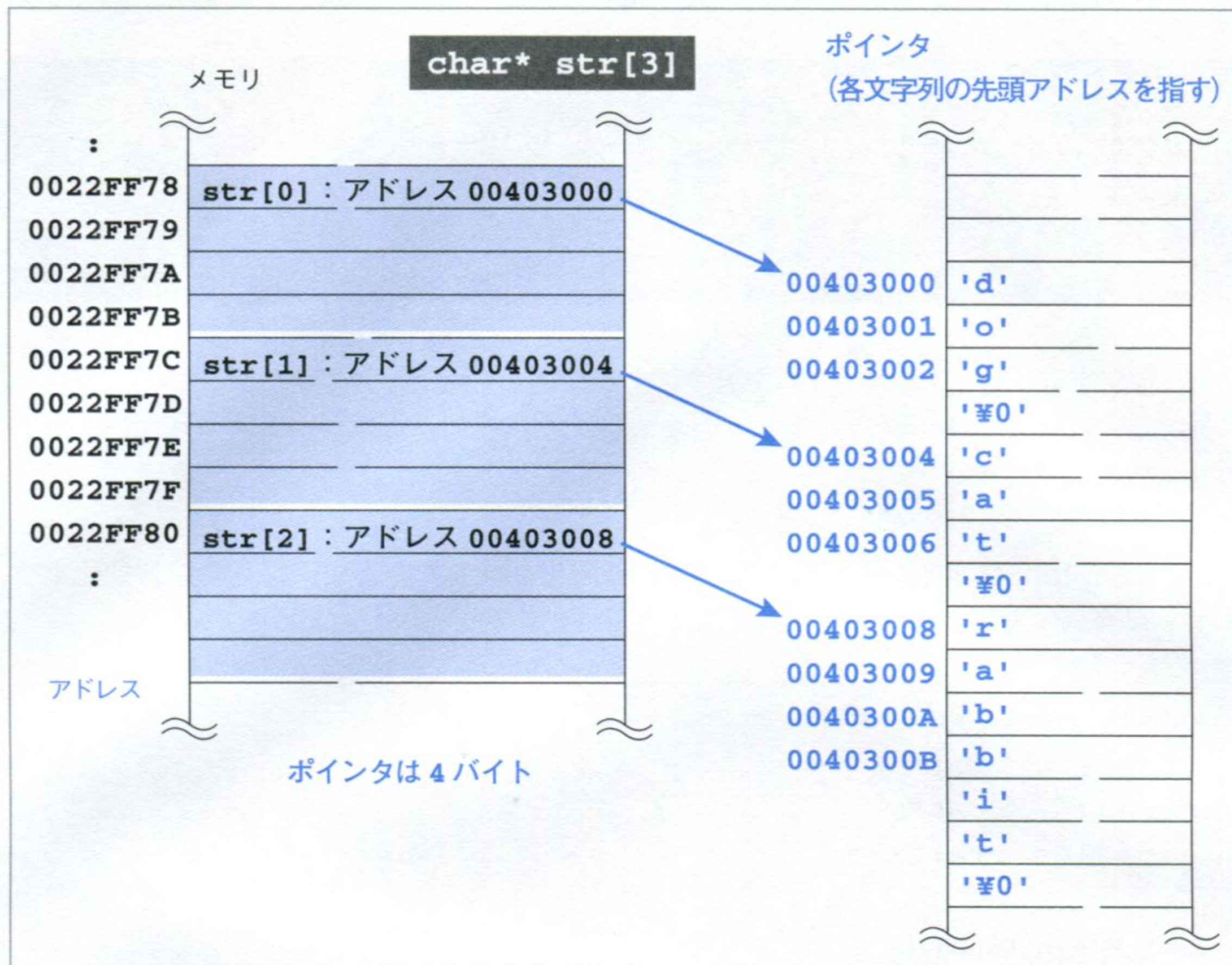
●ポインタ配列

char* str[3]

メモリ

ポインタ
（各文字列の先頭アドレスを指す）

アドレス	メモリ
:	
0022FF78	str[0]：アドレス 00403000
0022FF79	
0022FF7A	
0022FF7B	
0022FF7C	str[1]：アドレス 00403004
0022FF7D	
0022FF7E	
0022FF7F	
0022FF80	str[2]：アドレス 00403008
:	

アドレス	内容
00403000	'd'
00403001	'o'
00403002	'g'
	'¥0'
00403004	'c'
00403005	'a'
00403006	't'
	'¥0'
00403008	'r'
00403009	'a'
0040300A	'b'
0040300B	'b'
	'i'
	't'
	'¥0'

アドレス

ポインタは4バイト

ポインタは4バイト必要なので、ポインタがひとつずれるとアドレスは4つ先を示します。ポインタ配列の宣言は、他のポインタ変数と同様、

【ポインタ配列の宣言】

```
char* str[]
```

という書き方をします。簡単なプログラムで試してみましょう。

【5-2_sample4.c】

```
#include <stdio.h>

int main() {
    char* str[] = {"dog", "cat", "rabbit"};
    int i;

    for(i = 0; i < 3; i++) {
        printf("%p str[%d]: %s アドレス：%p¥n",
            str+i, i, str[i], *(str+i));
    }
    return 0;
}
```

```
0022FF78 str[0]: dog アドレス：00403000
0022FF7C str[1]: cat アドレス：00403004
0022FF80 str[2]: rabbit アドレス：00403008
    ↑                ↑                ↑
配列のアドレス  指しているアドレスの値  指しているアドレス
```

str[添え字番号]だけで、文字列を表すことができました。

ポインタ配列のアドレスは連続していますが、それぞれの文字列が格納されているアドレスは別の場所です。

ヒント

*7：文字列の代入にはイコール（=）を使わずにstrcpy関数を使いますが、ポインタ配列の各要素に格納するのはアドレスなので、str[0] = "dog"; と、イコールを使うことができます。

5-2_sample4.cはポインタ配列の宣言時に初期化を行っています。宣言時に初期化を行わない場合は、あとから値を設定しましょう。

```
char* str[3];
str[0] = "dog";*7
str[1] = "cat";
str[2] = "rabbit";
```

5 引数の受け取り

さて、ポインタ配列の書き方におぼえがないでしょうか？ 本日の1時限目の引数の受け取りに使った、

```
int main(int argc, char* argv[]) {
```

というのは、まさにポインタ配列です。プログラムが実行されると、プログラム名とその引数のデータはメモリ上のある位置に書き込まれます。ポインタ配列argvは、プログラム名とその引数の先頭データのアドレスを示すことになります。よって、argv[0]でプログラム名を、それに続いてargv[1]、argv[2]……で指定した引数を参照できるのです。

6 ポインタ配列のデータをランダム位置からすべて順に表示する

ここまで理解できたところで、この時限の冒頭で作成したプログラムを、よく見直してください。

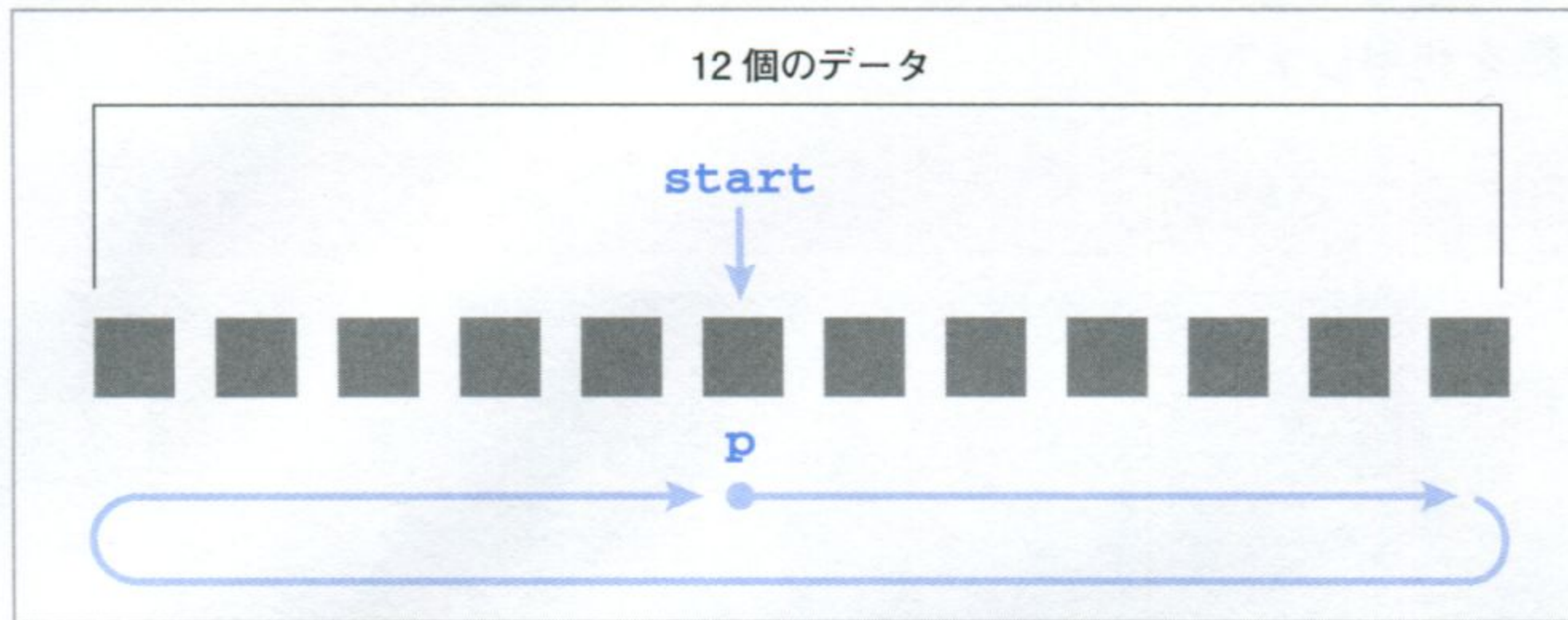
12個のデータを持つポインタ配列dataの要素を、すべて順に表示しましょう。表示をはじめる位置は、ランダムに設定します。

「現在どのデータを表示するか」の位置は、最初はランダムにきまった表示位置となります。そこから順にデータを表示するには、その位置を順にずらしていきます。その表示位置は、変数pを利用します。表示を行うポインタ配列のアドレスは、「data + p」になります。pの初期値はランダム値で、そこからひとつずつ位置をずらし、配列の最後になったら配列の頭に戻るようにします。

●12個のデータのうち、pの位置はランダムにきめる

12個のデータ

start

p

```
char *data[] = { (略：12個のデータ) };
int data_len = 12;
int i, p, start;*8

srand(time(NULL));
start = rand()%data_len;// スタート位置の決定
for(p = start, i = 1; i <= data_len; i++) {
        データの表示 ;
        p++;
        if(p == data_len) { p = 0; }
}
```

ヒント

*8：変数pは、ポインタ変数ではなくただのint型変数です。表示するデータの位置を表しています。

なお、配列のアドレスとその値が指し示しているアドレスは、

```
printf("%p %p¥n", data+p, *(data+p));
```

で確認できます。

まとめ

難しいと思われがちなポインタですが、実際のところ、「ポインタ＝指し示すもの」と理解していれば、結構単純な話です。

メモリとアドレスの話はコンピュータのしくみにもかかわってくるので、このあたりの話が理解できていると、将来大規模なプログラムを書くときにも、きっと役に立つでしょう。

第5日 3時限目

【山手線ゲームを作ろう③】

山手線ゲームを完成させよう

ポインタを利用した山手線ゲームを作ります。
ゲームを外回りに進めたい場合は、1時限目で学習した、プログラム実行時の引数を利用します。

今回作成する例題

```
コマンド プロンプト
C:¥source>yamanotesen.exe
古今東西山手線ゲ～～ム！
お題：星座の名前
コンピュータの番　ちゃん　ちゃん！ > おとめ
  プレイヤーの番　ちゃん　ちゃん！ > てんびん
コンピュータの番　ちゃん　ちゃん！ > さそり
  プレイヤーの番　ちゃん　ちゃん！ > いて
コンピュータの番　ちゃん　ちゃん！ > やぎ
  プレイヤーの番　ちゃん　ちゃん！ > みずがめ
コンピュータの番　ちゃん　ちゃん！ > うお
  プレイヤーの番　ちゃん　ちゃん！ > おひつじ
コンピュータの番　ちゃん　ちゃん！ > おうし
  プレイヤーの番　ちゃん　ちゃん！ > ふたご
コンピュータの番　ちゃん　ちゃん！ > かに
  プレイヤーの番　ちゃん　ちゃん！ > しし
あなたの勝ち！
C:¥source>
```

コンピュータとプレイヤーで山手線ゲームを行い、データをすべて出し終えたら、「あなたの勝ち！」と表示される

サンプルファイルはこちら 10days_c ▶ day05-03 ▶ yamanotesen.c

●このレッスンのねらい

山手線ゲームはコンピュータとプレイヤーが交互にデータを出しあうゲームです。
2時限目までで学習した内容をもとに、対戦型ゲームを完成させましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

int main(int argc, char* argv[]) {
    char *data[] = { "おひつじ", "おうし", "ふたご", "かに",
                     "しし", "おとめ", "てんびん", "さそり",
                     "いて", "やぎ", "みずがめ", "うお" };
    int data_len = 12;      //データの数
    char input[10];         //プレイヤー入力値
    int i, p, start;
    int win = 1;            //プレイヤーが勝ちの場合は1
    int mawari = 1;         //内回りの場合は1
    int p_turn = 0;         //プレイヤーのターンの場合は1

    if ((argc > 1) && (strcmp(argv[1], "-soto") == 0)) {
        mawari = 0;
    }

    srand(time(NULL));
    start = rand()%data_len;
    printf("古今東西山手線ゲ～～ム！¥n");
    printf("お題：星座の名前¥n");
    for(p = start, i = 1; i <= data_len; i++, p_turn = !p_turn) {
        Sleep(500);
        if(p_turn == 0) { printf("コンピュータ"); }
        else { printf(" プレイヤー"); }
        printf("の番　ちゃん　");
        Sleep(500);
        printf("ちゃん！ > ");
```

```
        if(p_turn == 0) {
            printf("%s¥n", *(data+p));
        } else {
            scanf("%s", input);
            while (getchar() != '¥n') {  }
            if(strcmp(*(data+p), input) != 0) {
                win = 0;
                break;
            }
        }

        if(mawari == 1) { p++; } else { p--; }
        if((mawari == 1) && (p == data_len)) { p = 0; }
        if((mawari == 0) && (p == -1)) { p = data_len-1; }
    }
    if(win == 1) { printf(" あなたの勝ち！ "); }
    else { printf(" あなたの負け！ "); }
    return 0;
}
```

ヒント

＊1：拡張子に注意して保存しましょう。

2 入力できたら、「yamanotesen.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、yamanotesen.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o yamanotesen yamanotesen.c
```

4 プログラムを実行する

```
C:¥source>yamanotesen.exe
```

一度も間違えずにデータを全て出しおえれば、プレイヤーの勝ち！

```
古今東西山手線ゲ～～ム！
お題：星座の名前
コンピュータの番　ちゃん　ちゃん！　>　ふたご
（略）
コンピュータの番　ちゃん　ちゃん！　>　おひつじ
　プレイヤーの番　ちゃん　ちゃん！　>　おうし
あなたの勝ち！
```

入力を間違えたら、プレイヤーの負け！ ゲーム終了！

```
C:¥source>yamanotesen.exe -soto
古今東西山手線ゲ～～ム！
お題：星座の名前
コンピュータの番　ちゃん　ちゃん！　>　うお
　プレイヤーの番　ちゃん　ちゃん！　>　おひつじ
あなたの負け！
```

解説

1 対戦型のゲーム

対戦型ゲームではプレイヤーの番とコンピュータの番が交互に入れ替わります。12個のデータを交互に表示するプログラムを考えてみましょう。「データを表示する」処理は同じなので、繰り返し処理の部分を、「コンピュータの番」と「プレイヤーの番」がそれぞれ行われるようにします。

p_turnというint型変数をつくり、その値が1のときはプレイヤーのターンで、0のときはコンピュータのターンとします。

```
int p_turn = 0; // プレイヤーのターンの場合は 1

for( 初期値 ; 終了条件 ; p_turn = !p_turn) {
    if(p_turn == 0) { コンピュータの番の処理 ; }
    else { プレイヤーの番の処理 ); }
    共通の処理 ;
}
```

繰り返しブロックの中で、コンピュータとプレイヤーで処理内容が異なる場合は、フラグを利用します。

「フラグを立てる」「フラグを降ろす」という言葉を聞いたことがある方もいるでしょう。フラグとは、そのまま「旗」の意味です。

フラグが立つとは、ある事象がONになっている状態のことで、逆にフラグを降ろすとは、その事象がOFFになっている状態のことです。コンピュータの世界では一般的に、ONのときを1、OFFのときを0とみなします。

このプログラムではp_turnをフラグとします。プレイヤーの番のときが1なので、プレイヤーの番のときは「フラグが立っている」状態になります。

●フラグが立っているときが、プレイヤーの番

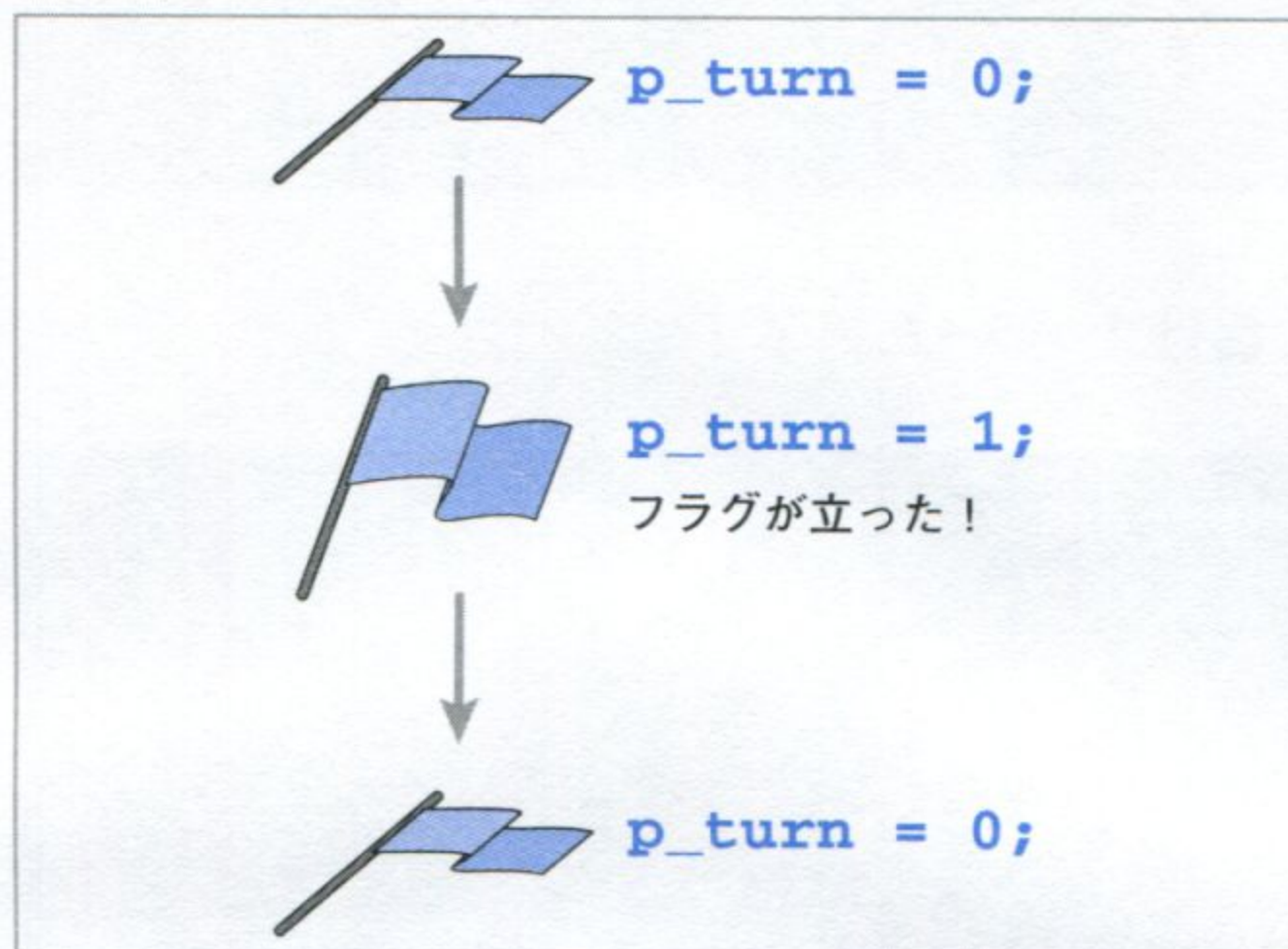
```
p_turn = 0;
```

↓

```
p_turn = 1;
```

フラグが立った！

↓

```
p_turn = 0;
```

繰り返し処理が終了したら、ターンをひっくり返します。「!p_turn」で現在のp_turn値をひっくり返し、それをまたp_turnに代入します。

これを繰り返すことにより、交互に処理を行う対戦型ゲームができあがります、

2 「待ち」時間を作る

山手線ゲームでは、コンピュータとプレイヤーの番が交互に来ます。プレイヤーの番のときは文字列を入力して［Enter］キーを押すので、その間、多少の時間がかかります。ですが、コンピュータの番のときは高速で処理が行われるため、すぐにデータが表示されてしまいます。それでは山手線ゲームとしての雰囲気が出ないので、コンピュータ側にも多少の「待ち」時間を持たせるようにします。

Sleep関数を使って、1秒だけ処理を待たせてみましょう。

【Sleep関数】

```
Sleep(1000);
```

とすると、1秒だけ処理を眠らせることができます。引数に秒数（単位：ミリ秒[*2]）を指定するので、1秒だと長いと感じる場合は、引数に500といった値を指定するとよいでしょう。Sleep関数を使うには、windows.hをインクルードします。

では、先ほど学習した対戦型のしくみも一緒に考えたプログラムを作成してみます。すべて自動で表示されるプログラムです。

ヒント

*2：UNIX処理系には、sleepという関数が存在します。こちらのsleep関数は、指定時間の単位は秒です。もしも使う機会があったら、引数の単位を間違えないようにしましょう。

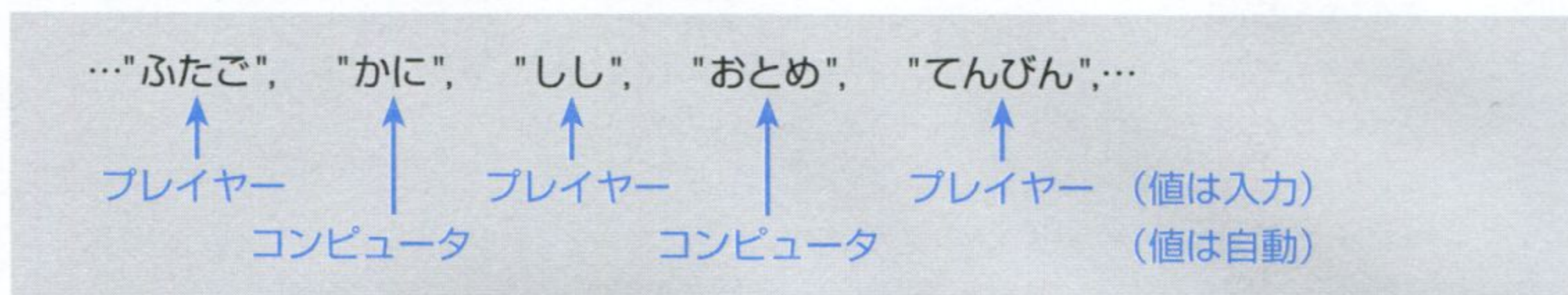
【5-3_sample1.c】

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <windows.h>

int main() {
    char *data[] = { （略：day05-02¥5-2.c 参照） };
    int data_len = 12;  // データの数
    int i, p, start;
    int p_turn = 0; // プレイヤーのターンの場合は 1

    srand(time(NULL));
    start = rand()%data_len;
    for(p = start, i = 1; i <= data_len; i++, p_turn = !p_turn) {
        Sleep(500);
        if(p_turn == 0) { printf("コンピュータ"); }
        else { printf("  プレイヤー"); }
        printf("の番 %s¥n", *(data+p));
        p++;
        if(p == data_len) { p = 0; }
    }
    return 0;
}
```

```
コンピュータの番 おとめ
  プレイヤーの番 てんびん
コンピュータの番 さそり
  プレイヤーの番 いて
コンピュータの番 やぎ
  プレイヤーの番 みずがめ
コンピュータの番 うお
  プレイヤーの番 おひつじ
コンピュータの番 おうし
  プレイヤーの番 ふたご
コンピュータの番 かに
  プレイヤーの番 しし
```

お互いのターンになったら、最初に0.5秒の待ち時間を入れています。なんとなく、ゲームの雰囲気が出てきたでしょうか。

3 山手線ゲームを完成させる

コンピュータ相手に山手線ゲームを行う場合、プレイヤーは値を入力し、コンピュータは自動で該当するデータを引き出します。

まず、2時限目に作ったプログラムを対戦型に作り直します。

今現在、どのデータが表示（判定）対象になっているかは、変数pを利用します。pの初期値はランダムで、プレイヤーまたはコンピュータの各ターンがおわったら、次の場所へ移動します。

それぞれのターンの処理を考えてみましょう。

コンピュータのターンのときは内容を表示するだけです。プレイヤーのターンのときは、値を標準入力から受け取り、それを現在位置のデータと一致するかどうか判定します。一致しない場合はプレイヤーの負けとなるので、繰り返し処理を強制終了します。

【それぞれのターンの処理：2時限目のプログラムを対戦型に改造】

```
for(p = start, i = 1; i <= data_len; i++, p_turn = !p_turn) {
  (略)

    if(p_turn == 0) {    // コンピュータのターンの処理
        printf("%s¥n", *(data+p));
    } else {             // プレイヤーのターンの処理
        値を入力;
        if( 値が *(data+p) の内容と一致しない場合 ) {
            間違えたのでプレイヤーの負け処理;
            break;
        }
    }
    p++;
    if(p == data_len) { p = 0; }
}
```

これに、1時限目で作成した外回り（配列前方向順）の機能をつけ加えます。

【外周り機能を追加：１時限目で作成したプログラムより】

```
int mawari = 1;      // 内回りの場合は1
int data_len = 12;   // データの数

if ((argc > 1) && (strcmp(argv[1], "-soto") == 0)) {
    mawari = 0;
}

for(p = start, i = 1; i <= data_len; i++, p_turn = !p_turn) {
(略：それぞれのターンの処理)

    if(mawari == 1) { p++; } else { p--; }
    if((mawari == 1) && (p == data_len)) { p = 0; }
    if((mawari == 0) && (p == -1)) { p = data_len-1; }
}
```

内回りのときは、配列の最終データの次は最初に戻ります。外回りのときは、配列の最初のデータの次は、最後のデータに戻ります。

```
          char *data[] = { "1", "2", "3", "4", "5" };
内回り          →  →  →  →  →  "1"に戻る
外回り          "5"に戻る ←  ←  ←  ←  ←
```

この配列の場合、データ"3"からはじまる内周りの場合は「"3"→"4"→"5"→"1"→"2"」、外周りの場合は「"3"→"2"→"1"→"5"→"4"」となります。

外回り、内回りともに次のデータ位置が配列外になってしまう場合に位置を修正すれば、すべてのデータを参照することができます。

あとは、最後にゲームの勝敗を表示し、Sleep関数をうまく利用してよりゲームらしくなるよう、表示を工夫すればできあがりです。

ゲームのお題は自由に変えることができます[*3]。その場合、データの内容とデータ数、お題の出力内容を変更してください。

> **ヒント**
> *3：付属CD-ROMに収録したソースコードには、山手線の駅データをコメントとして記述してあります。それを使う場合は、data配列を星座のデータと入れ替え、変数data_lenの値を29に変更してください。

まとめ

C言語で文字列の配列を作りたい場合はポインタ配列を使います。敬遠されがちな「ポインタ」ですが、ポインタ配列はあまり「ポインタ」とは意識せずに、単純に文字列を配列として扱うのに便利な概念として利用しましょう。

練習問題

Q

データの順番と方向性なしの山手線ゲームを作りなさい。

[条件]
ゲームの最初はコンピュータの番とし、データをすべて出しおわればプレイヤーの勝ち、データにない値を入力したときはプレイヤーの負けとする。

解答は巻末に

第 6 日

いつどこでゲームを作ろう

1時限目　ファイルの入出力について学ぼう①

2時限目　ファイルの入出力について学ぼう②

3時限目　いつどこでゲームを完成させよう

本日のレッスンでは、「いつ」「何処で」「誰と」「誰が」「何を」「どうした」をあらかじめ考えておき、書いた紙を順番に読んでいくという「いつどこでゲーム」を作成します。
C言語では、プログラムからファイルへの書き出しや読み込みを行うための関数が用意されています。それを利用すれば、各ゲームの結果を記録しておき、ランキングを出すことも可能になります。
1、2時限目ではファイルへの入出力の基本と、読み込み／書き出し関数を学び、ゲームのデータファイルを作成しておきます。3時限目で読み込んだデータファイルを使って、いつどこでゲームを完成させます。

今日作るプログラムについて

いつどこでゲーム

「いつどこでゲーム」とは、「いつ」「何処で」「誰と」「誰が」「何を」「どうした」をあらかじめ考えておき、書いた紙を順番に読んでいくというゲームです。
それぞれの内容を書いたのは別の人なので、組みあわせによってはかなり面白い内容になったりします。
最近ではインターネット上で複数の人が自由にデータを入力し、それをランダムに組みあわせて文書を作るゲームも見かけます。作り出す文書の内容も、物語の次回予告だったり、小説のあらすじだったり、さまざまです。

今回は、小説本の帯に書くキャッチコピーを、「いつどこでゲーム」で作ってみたいと思います。本の帯とは、本の表紙に巻いてある細長い紙のことで、本書だったら、

『「昨日よりできる」が実感できる！』

などと書いてある紙です。その本のキーワードや簡単なあらすじなどを書き、読者に興味を抱かせ、購買意欲を刺激するためにあります。

今回作成する「いつどこでゲーム」で表示する内容は、「本文中のセリフ」「誰々が」「何々した」「小説の種類」にします。

【テンプレート】
『「(セリフ)」
(人) が (○○した)、今話題の (種類) 小説』
【例】
『「こ、これが究極のおにぎりか！！」
グルメ　が　うなった、今話題の　グルメ　小説』

ゲームのやりかたは次のとおりで、表示は何度も繰り返して行えるようにします。

①データはファイルに保存
②データは手動でもプログラムからでも追加可能
③ゲームは、データファイルを読み込んでそれぞれのデータからランダムに選び、組みあわせて表示

いつどこでゲームの実際の動作

●データファイルへのデータ追加

1 データ追加プログラムを実行し、追加するデータの種類を入力して [Enter] キーを押す

```
C:¥source>itudoko_data.exe
『「(1)」(2) が (3)、今話題の (4) 小説』
1: セリフ 2: 人 3: 行動 4: ジャンル (5: 終了) データ種類は? > 4
```

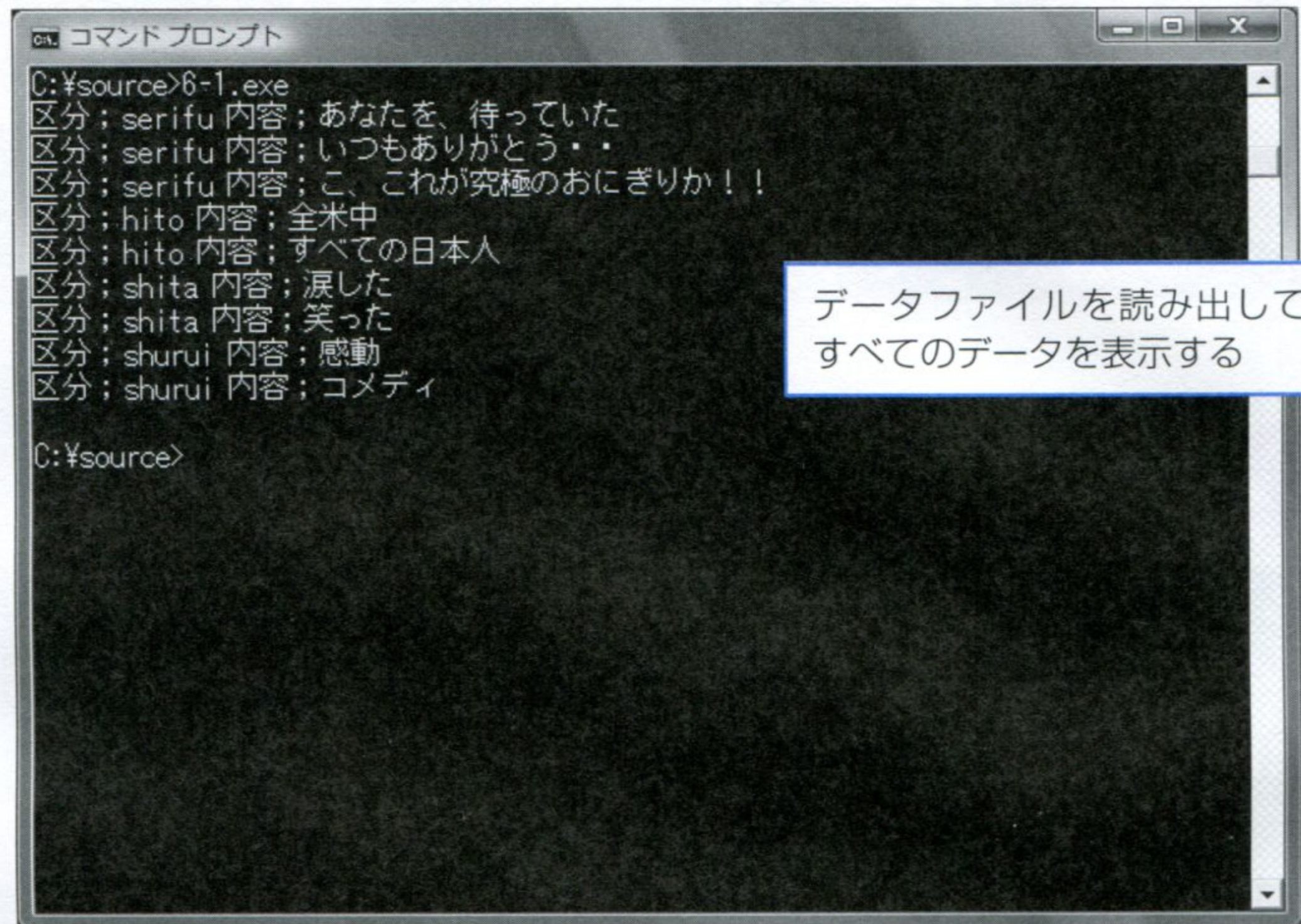
2 次に内容（254文字以内）を入力して、[Enter] キーを押す

```
C:¥source>itudoko_data.exe
『「(1)」(2) が (3)、今話題の (4) 小説』
1: セリフ 2: 人 3: 行動 4: ジャンル (5: 終了) データ種類は? > 4
内容は? > 携帯
```

3 追加したいデータの分だけ、手順①と②を繰り返す。データ追加を終了したい場合は、5を入力する

●いつどこでゲームの実行

1 いつどこでゲームを実行すると、データファイルのデータをもとに本の帯の文書が表示される

```
C:¥source>itudoko.exe

おすすめの１冊・・・
『「いつもありがとう・・」
全米中が笑った、今話題の携帯小説』

続ける：Enter　終了：Ctrl+z Enter >
```

2 続けて表示したい場合は、[Enter] キーを押す。表示を終了したい場合は、[Ctrl] + [z] キーと [Enter] キーを押す

第6日 1時限目 ファイルの入出力について学ぼう①

【いつどこでゲームを作ろう①】

ここではファイルの入出力の基本と、テキストファイルの内容の読み込み／書き出し方法を学びます。

今回作成する例題

```
C:¥source>6-1.exe
区分；serifu 内容；あなたを、待っていた
区分；serifu 内容；いつもありがとう・・
区分；serifu 内容；こ、これが究極のおにぎりか！！
区分；hito 内容；全米中
区分；hito 内容；すべての日本人
区分；shita 内容；涙した
区分；shita 内容；笑った
区分；shurui 内容；感動
区分；shurui 内容；コメディ

C:¥source>
```

データファイルを読み出して、すべてのデータを表示する

サンプルファイルはこちら 6days_c ▶ day06-01 ▶ 6-1.c

●このレッスンのねらい

通常、ファイルは、テキストエディタやその他の専用ソフトから作成したり、内容を見たり、コピーしたりできます。
C言語では、ファイルを扱う関数が用意されているので、プログラムからでもファイルの中身を読んだり、ファイルを作成したりすることができます。
ファイルの中身を読み込むことをファイル入力、ファイルへデータを書き出すことをファイル出力といいます。
この時限では、ゲームのデータファイルを読み出すプログラムを作成します。

プログラムを作成する

1 本書付属CD-ROM内の「day06-01」ディレクトリから、itudoko.txtを「C:¥source」ディレクトリ下にコピーする

●参考：itudoko.txtの内容

```
serifu	あなたを、待っていた
serifu	いつもありがとう・・
serifu	こ、これが究極のおにぎりか！！
hito	全米中
hito	すべての日本人
shita	涙した
shita	笑った
shurui	感動
shurui	コメディ
```

2 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    FILE *fp; // 入力ファイルのファイルポインタ
    char datafile[] = "itudoko.txt";
    char kubun[8], naiyou[256];
    char buff[512]; // 読み込んだ文字列を格納する

    // 入力ファイルをオープンする
    if((fp = fopen(datafile, "r")) == NULL) {
        printf("ファイルオープンエラー¥n");
        exit(1);
    }
    while(fgets(buff, sizeof buff, fp) != NULL){
        sscanf(buff, "%s¥t%s", kubun, naiyou);
        printf("区分；%s 内容；%s¥n", kubun, naiyou);
    }
    fclose(fp);

    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

3 入力できたら、「6-1.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

4 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、6-1.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 6-1 6-1.c
```

5 プログラムを実行する。itudoko.txtファイルの内容が表示されれば成功！

```
C:¥source>6-1.exe
```

▼

```
区分：serifu　内容：あなたを、待っていた
区分：serifu　内容：いつもありがとう・・
区分：serifu　内容：こ、これが究極のおにぎりか！！
区分：hito　　内容：全米中
区分：hito　　内容：すべての日本人
区分：shita　 内容：涙した
区分：shita　 内容：笑った
区分：shurui　内容：感動
区分：shurui　内容：コメディ
```

解説

1 ファイル入出力の基本

プログラムでファイルの入出力を行うためには、手順があります。ファイルからのデータの読み込みや書き出しを行うには、ファイルポインタを利用します。

ファイルポインタとは「ポインタ」、つまりファイルの中身を「指し示す」ものです。①入出力を行いたいファイルとファイルポインタを関連づけ、②ファイルへの操作は関連づけたファイルポインタを通じて行い、③ファイルへの操作が終了したら、ファイルポインタとの関連づけを切断します。

これがファイル入出力の手順です。順に詳しく見ていきましょう。

(1) ファイルポインタの宣言

最初に、ファイルと関連づけるためのファイルポインタを宣言します。

【ファイルポインタの宣言】

```
FILE *fp;
```

型：FILE　ファイルポインタ名：fp

ファイルポインタのデータ型はFILE型で、ファイルポインタ名は変数名と同様、好きにつけることができます。ファイルポインタの「ポインタ」とは、第5日で学習したあのポインタのことです。ファイルポインタはファイルの中の位置を指し示しています。

(2) ファイルとファイルポインタを関連づける

操作を行いたいファイルと、先ほど宣言したファイルポインタを関連づけます。この作業を、ファイルを開く（オープンする）といいます。

ファイルをオープンするには、fopen関数[*2]を使います。この作業を行ったあと、ファイル「test.txt」へのアクセスは、ファイルポインタfpを使って行うことができます。

ヒント
*2：この章で出てくるファイル入出力関連の関数は、すべてstdio.hをインクルードします。

【fopen関数によるファイルとファイルポインタの関連づけ】

```
fp = fopen("test.txt", "r");
```

ファイルポインタ：fp　ファイル名："test.txt"　オープンモード："r"

fopen関数の第1引数には扱うファイル名を指定し、第2引数にはそのファイルを「読み込み専用」で扱うか「書き出し専用」で扱うかなどを指定します。この指定をファイルオープンモードと呼びます。

●ファイルオープンモード一覧

モード	意味
r	読み込み専用
w	書き出し専用（ファイルあれば内容クリア）
a	追加書き出し専用
r+	読み書き両用（ファイルは存在済）
w+	読み書き両用（ファイルあれば内容クリア）
a+	読み書き両用で追加／作成する
rb	バイナリモード[*3]で読み込み専用
wb	バイナリモード[*3]で書き出し専用
ab	バイナリモード[*3]で追加書き出し専用
rb+ / r+b	バイナリモード[*3]で読み書き両用
wb+ / w+b	バイナリモード[*3]で読み書き両用
ab+ / a+b	バイナリモード[*3]で読み書き両用で追加／作成

ヒント
*3：バイナリモードとは、バイナリファイルを扱うときの処理のことです。バイナリファイルについては、本日の2時限目に学習します。

ファイルポインタの関連づけがおわったら、あとは自由にファイルにアクセスできます。しかし、fopen関数の実行時に読み込み専用でファイルをオープンした場合は読み込み作業のみ、書き出し専用でファイルをオープンした場合は書き出し作業のみを行います。読み書きのための関数は何種類か用意されているので、あとで紹介します。

(3) ファイルとファイルポインタの関連づけを切断する

ファイルの読み込み／書き出しが終了し、プログラム中でそのファイルポインタをもう利用しない場合、fclose関数を使ってファイルポインタを閉じる、つまりファイルポインタとファイルとの関連づけを切って、ファイルを閉じます。これを、ファイルをクローズするともいいます。

【fclose関数】

```
fclose(fp);
```


この作業を行うと、そのファイルポインタを使ってファイルにアクセスすることはできなくなります。もう一度アクセスしたいときは、再びfopen関数を使います。

まとめると、プログラムでファイルを扱うときは、

①ファイルポインタの用意
②ファイルのオープン
③読み書き作業
④ファイルのクローズ

という流れで行います。

fopen関数を使ってファイルを開き、fclose関数を使ってファイルを閉じます。その間に行う作業のための関数を、次から紹介します。

2 ファイルの指定方法

ファイルの読み込み／書き出し関数を紹介する前に、ファイル名の指定方法について補足しておきます。

ファイルを扱うには、fopen関数でオープンするファイル名を指定しました。

```
fp = fopen("test.txt", "r");
```

このときのファイル名の指定方法は、第1日に学習した絶対パスか、またはカレントディレクトリから見た相対パスで指定します。

(1) 絶対パス指定

ファイル名の指定をダブルクォートで括っているため、ディレクトリの区切りを表す記

号「¥」は、「¥¥」と表記[*4]しなければなりません。

```
fp = fopen("C:¥¥source¥¥test.txt", "r");
```

> **ヒント**
> *4：ダブルクォートの中で「¥」を使うと、「¥」は特殊文字の一部とみなされます。

(2) 相対パス指定

相対パス指定では、カレントディレクトリから見たパスを指定します。

つまり、プログラムを実行するカレントディレクトリにfopen関数でオープンしたいファイルがあるなら、そのファイル名だけを指定します。

```
fp = fopen("test.txt", "r");
fp = fopen("data¥¥test.txt", "r");
```

例えば、ファイル構成が次のような場合、

```
C:¥source¥
├ 6-1_sample.exe    プログラムファイル
└ test.txt          プログラムの中からオープンするファイル
```

C:¥sourceディレクトリから実行すると、目的のファイルがオープンできます。

```
C:¥source>6-1_sample.exe
```

しかし、C:¥がカレントディレクトリのとき[*5]に実行ファイルを相対パス指定で実行した場合は、

> **ヒント**
> *5：カレントディレクトリが「C:¥」のとき、sourceディレクトリの6-1_sample.exeを実行するには、絶対パス指定か、「source¥6-1_sample.exe」と相対パス指定で行います。

```
C:¥>source¥6-1_sample.exe
```

C:¥test.txtが存在しなければオープンエラーとなり、C:¥test.txtが存在すれば、そちらのファイルを開いてしまいます。

3 ファイル入力関数ー文字単位

まずはファイルの入力、つまり読み込みを行う関数について説明します。

(1) ファイルを1文字ずつ読み込むfgetc関数

ファイルへの操作は、fopen関数で関連づけたファイルポインタを使って行います。ファイルからの読み込みは、fgetc関数を使って1文字ずつ行うことができます。

【fgetc関数】

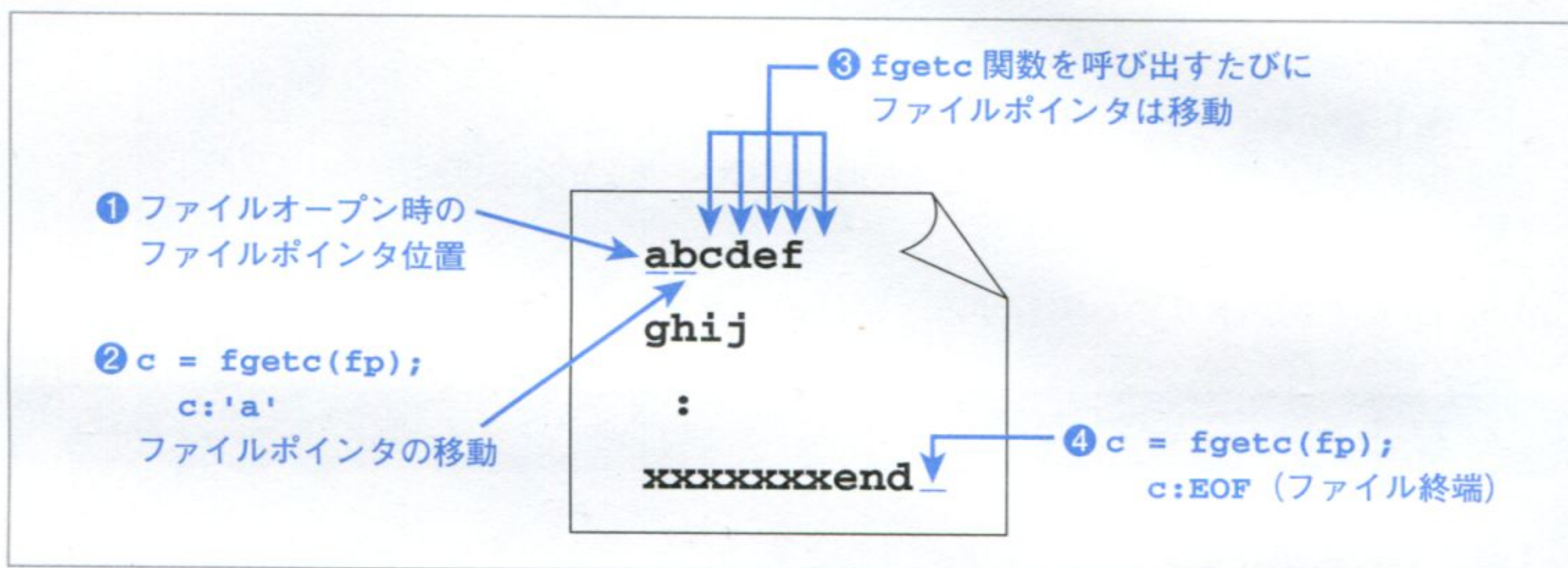
```
fgetc( ファイルポインタ )
```

fgetc関数の引数にファイルポインタを指定すると、関連づけられたファイルポインタから1文字ずつ読み込みます。成功すると読み込んだ文字を返し、エラーが発生するか、またはファイルの終端に達するとEOFを返します。EOFとはEnd Of Fileといって、ファイルのおわりを表します。

正確には、fgetc関数は文字をunsigned char型として読み、それをint型にキャスト、つまり一時的にint型への変換を行って返します。よって、読み込んだ値はint型で受け取りましょう[*6]。

ファイルポインタは、ファイルの中の読み込み／書き出し位置を指定するものです。ファイルをオープンすると、ファイルポインタはファイルの先頭を指しています。fgetc関数を使うと1文字読み込まれ、ファイルポインタは次の文字のところへ移動します。fgetc関数で読み込みを行うたびに移動するので、読み込みを続けてファイルの最後までくれば、EOFを返します。そこから先は読み込むことはできません。

ヒント
*6：キャストは第4日2時限目にも学習しました。int型で受け取る理由の詳細は、本日の3時限目のおわりに説明します。

●ファイルの読み込み

fgetc関数を使ってファイルの中身を最初から最後まで読み込んで、標準出力するだけのプログラムを作ってみましょう[*7]。

最初に、fopen関数でファイルを開きます。ファイルは読み込み専用で開きます。fopen関数は、ファイルのオープンに成功するとファイルポインタを返しますが、このときにNULLが返ってくると、ファイルが何らかの理由で開けなかった[*8]ということなので、これ以上作業する必要はありません。エラーとして処理しましょう。

ヒント
*7：ファイル入出力関数を利用するには、stdio.hをインクルードします。

ヒント
*8：ファイルを開けなかった理由として一番多いのは、ファイルが存在しない場合です。指定するファイル名の書き間違いに注意しましょう。

エラー処理の基本は、次の2点です。

・エラーの内容を出力表示する。
・プログラムをその場で強制終了する、exit関数[*9]を使う。引数には「異常終了」を表す「1」を指定し、exit(1)とする。

ヒント
*9：exit関数を利用するには、stdlib.hをインクルードします。

ファイルポインタが無事に取得できたら、そこから1文字ずつ読み込むだけです。ファイルの最後はEOFが返るので、それまでwhile文を使って読み込みを繰り返します。

読み込んだ文字はそのまま標準出力し、最後にfclose関数でファイルを閉じて終了します。

【6-1_sample1.c】

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fp; // ファイルポインタ
    int c; // 読み込んだ文字を格納する

    if((fp = fopen("6-1_sample1.c", "r")) == NULL) {
        // ファイルがオープンできなかった場合の処理
        printf(" ファイルオープンエラー ¥n");
        exit(1); // 強制終了する
    }

    // ファイルを読み込む
    while((c = fgetc(fp)) != EOF){
        printf("%c", c);
    }

    fclose(fp);
    return 0;
}
```

▼

```
コマンド プロンプト

C:¥source>6-1_sample1.exe
#include <stdio.h>
#include <stdlib.h>

int main() {
        FILE *fp; //ファイルポインタ
        int c; //読み込んだ文字を格納する

        if((fp = fopen("6-1_sample1.c", "r")) == NULL) {
                //ファイルがオープンできなかった場合の処理
                printf("ファイルオープンエラー¥n");
                exit(1); //強制終了する
        }

        //ファイルを読み込む
        while((c = fgetc(fp)) != EOF){
                printf("%c", c);
        }

        fclose(fp);
        return 0;
}

C:¥source>_
```

サンプルプログラムでは「6-1_sample1.c」ファイル自身を読み込んでいます。
読み込みは、while文の条件式の中で行っています。

```
while((c = fgetc(fp)) != EOF){
```

まず、c = fgetc(fp) でファイルから1文字読み出しています。読み出した文字が変数cに格納されるので、その値がEOFであるかどうか判断しています。EOFでないのならファイルは最後まで読み出していないので、処理を繰り返します。
このように、代入と同時に比較を行う条件式の書き方もおぼえておいてください。

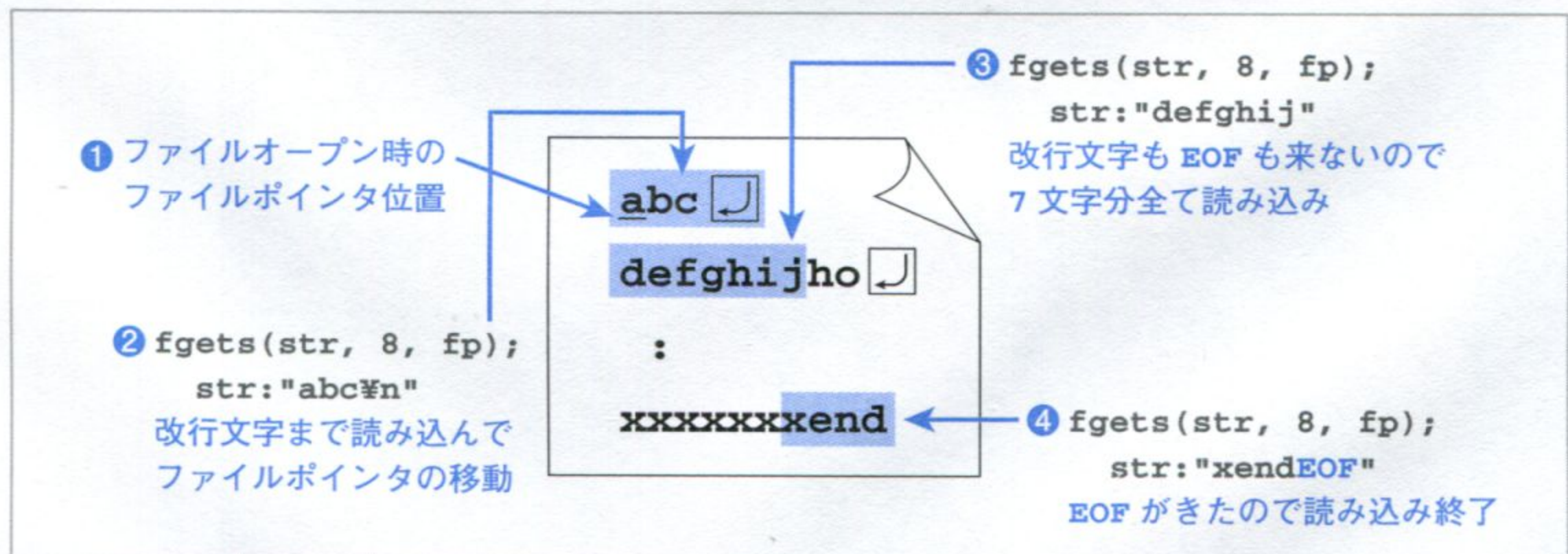
(2) ファイルをまとめて読み込むfgets関数

ファイルを1文字ずつ読み込むfgetc関数に対して、指定した分だけまとめて読み込むのがfgets関数です。

【fgets関数】

```
fgets(読み込んだ文字列を格納する変数, 読み込み文字数, ファイルポインタ)
```

ファイルポインタから、指定した「読み込み文字数－1」文字分を読み込んで、変数に格納します。途中で改行文字「¥n」が現れたら、読み込み最大文字数に達していなくても、その回の読み込みは終了します。読み込んだ変数の最後には文字列の最後を表す「¥0」がつきます。
成功すると読み込んだ文字列のポインタを返し、エラーが発生するかまたはファイルの終端であるEOFに達すると、NULLを返します。

●ファイルの読み込み

fgets関数を使ったプログラムを作成してみましょう。
文字列変数strに格納された読み込み文字列をそのまま標準出力していけば、ファイルの中身をすべて表示できます。

【6-1_sample2.c】

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fp; // ファイルポインタ
    char str[8]; // 読み込んだ文字列を格納する

    if((fp = fopen("6-1_sample2.c", "r")) == NULL) {
        printf("ファイルオープンエラー¥n");
        exit(1);

    }

    // ファイルを読み込む
    while(fgets(str, 8, fp)){
        printf("%s", str);
    }
    fclose(fp);
    return 0;
}
```

(3) 書式つきで読み込むfscanf関数

書式にしたがって読み込みを行うことができるのが、fscanf関数です。

【fscanf関数】

```
fscanf(ファイルポインタ, 書式, 格納変数)
```

ファイルポインタから、書式に従って読み込みを行い、指定された格納変数のアドレスに格納します。標準入力scanf関数のファイル版[*10]、と考えることができます。

```
int d;
fscanf(fp, "%d", &d);
```

ヒント

*10：scanf関数とfscanf関数は、読み込みデータが書式どおりでないとうまく使えない、という欠点があります。

4 ファイル出力関数－文字単位

次に、ファイルへの出力、つまり書き出しを行う関数[*11]について説明します。

> **ヒント**
> *11：第2日で紹介したfprintf関数も、ファイル出力関数です。第1引数にフィアルポインタを指定すると、ファイル出力できます。

（1）ファイルを１文字ずつ書き出すfputc関数

ファイルを1文字ずつ読み込むfgetc関数に対して、1文字ずつ書き出すのがfputc関数です。

【fputc関数】

```
fputc( 書き出す文字 , ファイルポインタ )
```

fputc関数を実行するたびに、ファイルポインタに1文字ずつ書き出します。1文字書き出すと、ファイルポインタは次の文字に移動します。

（2）ファイルをまとめて書き出すfputs関数

ファイルからまとめて読み込むfgets関数に対して、まとめて書き出すのがfputs関数です[*12]。

> **ヒント**
> *12：fgetc関数に対応するのがfputc関数、fgets関数に対応するのがfputs関数とおぼえておきましょう。

【fputs関数】

```
fputs( 書き出す文字列 , ファイルポインタ )
```

fputsを実行すると、ファイルポインタに文字列を書き出します。書き出すと、ファイルポインタは文字列分、移動します。

文字列の最後には「¥0」がありますが､それは出力されません。書き出しに成功すると､負でない値を返します。

5 データファイルを読み込んで表示するプログラム

今回は、あらかじめデータファイルを用意しておき、それを読み込むプログラムを作ります。読み込むデータファイル名をitudoko.txtとします。これは､いつどこでゲームプログラムで使用するデータファイルで、中身は次のように「区分」「内容」の1セットを1行にしています。2つの単語の間は、タブで区切ります。

【itudoko.txt】

```
[区分]   [内容]
serifu  あなたを、待っていた
serifu  いつもありがとう・・
serifu  こ、これが究極のおにぎりか！！
hito    全米中
hito    すべての日本人
(略)
```

このデータを、fgets関数を使って読み込みます。これは、先ほど作成した6-1_sample2.cのファイル名を変更すれば、すぐにできます。

今回はfgets関数を使って1行を1回で読み込み、sscanf関数を使って、それを「区分」と「内容」に分けてみましょう。読み込んだ1行は、変数buffに格納されます。

```
char buff[512];
```

1行あたりのデータの文字数には上限を設定してあるので、1回の読み込みで余裕をもって1行を読み込むことができます。

serifu	あなたを、待っていた
[区分]	[内容]
７文字以内	254文字以内

読み込みにはsscanf関数を使用します。sscanf関数は、文字列を書式に従って読み込み*13、指定された格納変数のアドレスに格納します。

【sscanf関数】

```
sscanf( 文字列 , 書式 , 格納変数 )
```

実際にsscanf関数使ってみましょう。

区分と内容を格納する配列をそれぞれ用意します。読み込んだ1行はすべて「区分(タブ)内容（改行)」という構成になっているので*14、sscanf関数の書き方に従って、buffをkubunとnaiyouに分割します。

```
char kubun[8], naiyou[256];

while(fgets(buff, sizeof buff, fp) != NULL){
    sscanf(buff, "%s¥t%s", kubun, naiyou);
    printf(" 区分；%s 内容；%s¥n", kubun, naiyou);
}
```

while文の条件式にあるsizeof*15は、変数の値を求める演算子です。sizeof buffで、buffの大きさ（バイト数）を算出します。sizeof(buff)と書いても同じです。

まとめ

ファイル入出力の基本と、読み込み／書き出しに使用する関数について学習しました。

これらは、今後のプログラムで使用するデータや結果を記録したりするのに必ず必要となる知識です。しっかりと理解できたでしょうか。

ヒント

*13：読み込み書式はscanf関数やfscanf関数と同様に指定します。sscanf関数は、fscanf関数の読み込み対象が文字列になったものです。

ヒント

*14：sscanf関数では、改行、タブは空白と同じ扱いです。空白は読み込みの区切りとして扱われます。よって、"%s¥t%s"は"%s %s"と指定しても同じです。

ヒント

*15：引数にはデータ型を指定することもできます。char型は1バイトなので、sizeof(char)の値は1になります。

第6日 2時限目

ファイルの入出力について学ぼう②

【いつどこでゲームを作ろう②】

バイナリファイルの入出力の学習と、いつどこでゲームのデータを追加するプログラムを作成します。

今回作成する例題

```
C:¥source>itudoko_data.exe
『「(1)」(2) が (3) 、今話題の (4) 小説』
1:セリフ 2:人 3:行動 4:ジャンル (5:終了) データ種類は？ > 2
内容は？ > 力士
データを追加しました
『「(1)」(2) が (3) 、今話題の (4) 小説』
1:セリフ 2:人 3:行動 4:ジャンル (5:終了) データ種類は？ > 3
内容は？ > 寄り切った
データを追加しました
『「(1)」(2) が (3) 、今話題の (4) 小説』
1:セリフ 2:人 3:行動 4:ジャンル (5:終了) データ種類は？ > 1
内容は？ > 良い寄りきりだ・・・
データを追加しました
『「(1)」(2) が (3) 、今話題の (4) 小説』
1:セリフ 2:人 3:行動 4:ジャンル (5:終了) データ種類は？ > 5

C:¥source>
```

いつどこでゲームのデータファイルにデータを追加する

サンプルファイルはこちら　10days_c ▶ day06-02 ▶ itudoko_data.c

●このレッスンのねらい

コンピュータ上で扱うファイルは、テキストファイルとバイナリファイルの2種類に分けられます。今までの学習で扱ってきたのは、すべてテキストファイルです。

この時限では、バイナリファイルの入出力について学習します。

この2時限目と3時限目で、いつどこでゲームのデータファイルにデータを追加するプログラムを作成します。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    FILE *fp; // 入力ファイルのファイルポインタ
    char datafile[] = "itudoko.txt";
    int k;
    char str[256];
    char *kubun[] = { "serifu", "hito", "shita", "shurui" };

    // 入力ファイルをオープンする
    if((fp = fopen(datafile, "a+")) == NULL) {
        printf("ファイルオープンエラー¥n");
        exit(1);
    }

    while(1) {
        printf("『「(1)」(2)が(3)、今話題の(4)小説』¥n");
        printf("1:セリフ 2:人 3:行動 4:ジャンル (5:終了)データ種類は？ > ");
        k = 0;
        scanf("%d", &k);
        while (getchar() != '¥n') { }
        if(k == 5) { break; }
        if((k < 1) || (k > 4)) {
            printf("1～5までの数値を入力してください¥n");
            continue;
        }
        printf("内容は？ > ");
        fgets(str, 256, stdin);
        if(str[0] == '¥n') {
            printf("内容を入力してください¥n");
            continue;
        }
        if((strlen(str) == 255) && (str[254] != '¥n')) {
            printf("文字数オーバーです¥n");
            while (getchar() != '¥n') { }
            continue;
        }
```

```
        fprintf(fp, "%s¥t%s", kubun[k-1], str);
        printf("データを追加しました¥n");
    }
    fclose(fp);
    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら「itudoko_data.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、itudoko_data.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o itudoko_data itudoko_data.c
```

4 プログラムを実行する

```
C:¥source>itudoko_data.exe
```

▼

```
『「(1)」(2) が (3)、今話題の (4) 小説』
1: セリフ 2: 人 3: 行動 4: ジャンル (5: 終了) データ種類は？ > 2
内容は？ > プロ棋士
データを追加しました
『「(1)」(2) が (3)、今話題の (4) 小説』
1: セリフ 2: 人 3: 行動 4: ジャンル (5: 終了) データ種類は？ > 4
内容は？ > 携帯
データを追加しました
『「(1)」(2) が (3)、今話題の (4) 小説』
1: セリフ 2: 人 3: 行動 4: ジャンル (5: 終了) データ種類は？ > 8
1～5までの数値を入力してください

『「(1)」(2) が (3)、今話題の (4) 小説』
1: セリフ 2: 人 3: 行動 4: ジャンル (5: 終了) データ種類は？ > 5
```

区分と内容のセットの入力を繰り返す

区分が1～4以外の場合、または内容が254文字より大きかったり内容がなかったりした場合は、入力エラーとして次の入力を行う

区分で5を入力したときに終了すれば成功！

解説

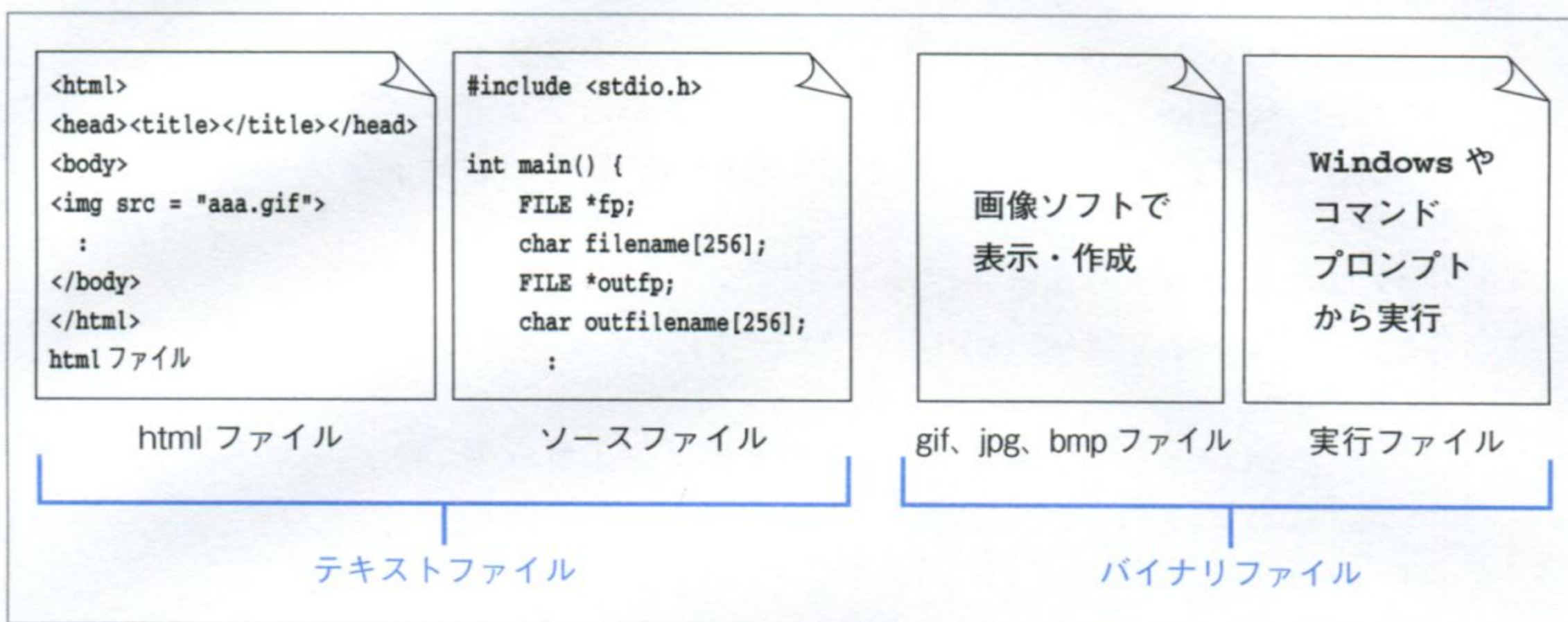
1 テキストファイルとバイナリファイル

今までの学習で扱ったファイルはすべてテキストファイルです。テキストファイルとは、目で見て理解できる、テキストエディタで見ることのできるファイルです。

ファイルにはもうひとつ、バイナリファイル[*2]という種類が存在します。こちらはテキストファイルとは逆に、ファイルを開いて見ても理解できません。

例えば、プログラムのソースファイルはテキストファイルです。そして、画像ファイルや、プログラムの実行ファイルがバイナリファイルです。

バイナリファイルを扱うには、fread関数とfwrite関数[*3]を使いましょう。

●テキストファイル、バイナリファイル

ヒント
*2：バイナリファイルは、一般的に個々の特別なソフトを使うことで利用できるファイルです。「バイナリ」とは2進法を意味します。つまり、2進数＝コンピュータが理解できる言語（機械語）で書かれたファイルです。
バイナリファイルのエディタソフトも存在します。

ヒント
*3：fread関数、fwrite関数を使用するには、stdio.hをインクルードします。

2 ファイル入力関数ーバイト単位

fread関数は、ファイルをバイト単位で読み込む関数です。

【fread関数】

```
fread( 読み込んだものを格納するバッファ , 読み込み単位（バイト）,
                          読み込み回数 , ファイルポインタ )
```

ファイルポインタから指定した読み込みバイト数ずつ指定回数分読み出して、バッファに格納します。fgets関数と似ていますが、改行文字などを区別しません。すべてのデータをひたすら黙々とバイト単位で読み込みます。

fread関数は、読み込んだ回数を返します。すべて読み込みおわると結果が0になるので、それまで繰り返し読み込みます。

3 ファイル出力関数ーバイト単位

ファイルをバイト単位で書き出す関数が、fwrite関数です。

【fwrite関数】

```
fwrite( 書き出すものを格納しているバッファ , 書き出し単位（バイト）,
                              書き出し回数 , ファイルポインタ )
```

ファイルポインタの場所に、指定した書き出しバイト数ずつ、指定回数分書き出します。ちょうどfread関数と逆の動作をします。

4 バイナリファイルのコピーを作成する

ファイルを読み込み、それを新規ファイルに書き出すプログラム、つまりコピーファイルを作成するプログラムを作成してみましょう。

【6-2_sample1.c】

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fp;          // 読み込みファイルのファイルポインタ
    FILE *wfp;         // 書き出しファイルのファイルポインタ
    char buff[8];      // 書き出すもの格納しているバッファ

    // 読み込みファイルオープン
    fp = fopen("6-1.exe", "rb");
    if (!fp){
        printf("読み込みファイルオープンエラー¥n");
        exit(1);
    }

    // 書き出しファイルオープン
    wfp = fopen("6-1_.exe", "wb");
    if (!wfp){
        printf("書き出しファイルオープンエラー¥n");
        exit(1);
    }

    // ファイルの読み込み＆書き出し
    while (fread(buff, sizeof(char), 8, fp) != 0) {
        fwrite(buff, sizeof(char), 8, wfp);
    }
    fclose(wfp);
    fclose(fp);
    return 0;
}
```

読み込み、書き出しのためのファイルポインタを2つ用意し、それぞれをバイナリモードでオープンします。

fread関数で読み込んだものをそのままfwrite関数で書き出して[*4]、ファイルに出力します。

ヒント

*4：fread関数、fwrite関数は、テキストファイルの読み書きにも使用することができます。6-2_sample1.cのプログラムでは、テキストファイルのコピーも作れます。

最後に両方のファイルをクローズしておわりです。

このプログラムでは、前の時限に作成した6-1.exeファイルをコピーしています。ファイル6-1.exeのコピー「6-1_.exe」ができていることを確認しましょう。もしもコピーに失敗していると、6-1_.exeファイルは実行することができません。

●6-1_.exeの実行結果

```
コマンド プロンプト
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:¥Users¥user>cd ¥source

C:¥source>gcc -o 6-2_sample1 6-2_sample1.c

C:¥source>6-2_sample1.exe

C:¥source>6-1_.exe
区分；serifu 内容；あなたを、待っていた
区分；serifu 内容；いつもありがとう・・
区分；serifu 内容；こ、これが究極のおにぎりか！！
区分；hito 内容；全米中
区分；hito 内容；すべての日本人
区分；shita 内容；涙した
区分；shita 内容；笑った
区分；shurui 内容；感動
区分；shurui 内容；コメディ

C:¥source>
```

5 いつどこでデータの追加

ファイル入出力についての説明は、fread関数とfwrite関数でおわりです*5。

では、いつどこでゲームのデータファイルにデータを追加するプログラムを作成してみましょう。

データはプログラムからではなく手動*6で作成・追加することもできるので、これはいつどこでゲームのおまけ的なプログラムになります。

(1) ファイルを追加書き出しモードでオープンする

いつどこでゲームのデータファイルであるitudoko.txtを追加書き出しモードでオープンします。もしitudoko.txtが存在しない場合は、新規作成します*7。

```
FILE *fp;
char datafile[] = "itudoko.txt";

if((fp = fopen(datafile, "a+")) == NULL) {
    printf("ファイルオープンエラー¥n");
    exit(1);
}
(略)
fclose(fp);
```

ヒント
*5：いつどこでゲームのデータファイルはテキストファイルなので、データの追加にはfprintf関数を使います。
fread関数、fwrite関数は使用しません。

ヒント
*6：データを手動で追加する場合は、データの書式と文字数制限を守って行ってください。
最後のデータ行には、必ず改行を入ることも忘れずに。

ヒント
*7：ファイルが存在しない場合にきちんと新規作成されるかどうか、別のファイル名で試してみてください。

(2) データを入力する

データは区分と内容の両方を、繰り返し入力します。区分の内訳は次のとおりです。

●区分の内訳

区分	データ種類
serifu	セリフ
hito	人
shita	行動
shurui	ジャンル

入力された内容をそれぞれチェックし、問題があればメッセージを表示して次の読み込みを行います。区分で5が入力されたら、データ追加を終了します。

なお、itudoko.txtがすでに存在している場合、データファイルの最後のデータ行が改行されていることを確認してください。もしもそうでない場合は、最後のデータ行に改行を入れてください。

内容の入力にはfgets関数を使い、第2引数に読み込む文字数、第3引数に読み込み先を指定しています。

ヒント
*8：255文字より多い場合はバッファに文字が残っているので、次の入力のためにクリアしておく必要があります。

ヒント
*9：読み込みは最後の改行も含めて255文字以内とします。読み込み文字数が255で、最後の文字str[254]が改行ならば問題ありません。

```
fgets(str, 256, stdin);
```

第3引数にあるstdinとは、標準入力を表す識別子です。

この場合、stdinつまり標準入力から最大「256－1」の文字数を読み込みます。

gets関数だと文字数を指定できず、scanf関数では文字数の指定に間違いがあった場合や予定よりも長い文字列が入力されたときに、プログラムに不具合が発生することがあります。これを防ぐためにfgets関数を使用しました。

読み込んだ文字列が改行を含めて255文字*8より多い場合は、str[254]が改行になっていないはずなので、文字数オーバーとします*9。

```
while(1) {
    printf("『「(1)」(2)が(3)、今話題の(4)小説』¥n");
    printf("1:セリフ 2:人 3:行動 4:ジャンル (5:終了)データ種類は？ > ");
    区分入力
        5が入力されたらbreak
        それ以外は値チェック。1～4以外ならcontinue
    内容入力
        文字数チェック
    "区分¥t内容"をファイルに書き込む
}
```

内容チェックを行った状態だと、文字列の最後には必ず¥nがついているので、出力は「区分¥t内容」のみです。

(3) ファイルポインタの移動

この時限の最後に、ファイルポインタの移動について説明しておきましょう。なお、ファイルポインタの移動はこの時限のプログラムでは使用しませんが、3時限目のおわりの練習問題で使用します。

ファイルをオープンしたときのファイルポインタの位置は、オープンモードがaの場合はファイルの末尾に、それ以外はファイルの先頭になります。

fseek関数を使って、ファイルポインタの位置を移動することができます。

【fseek関数】

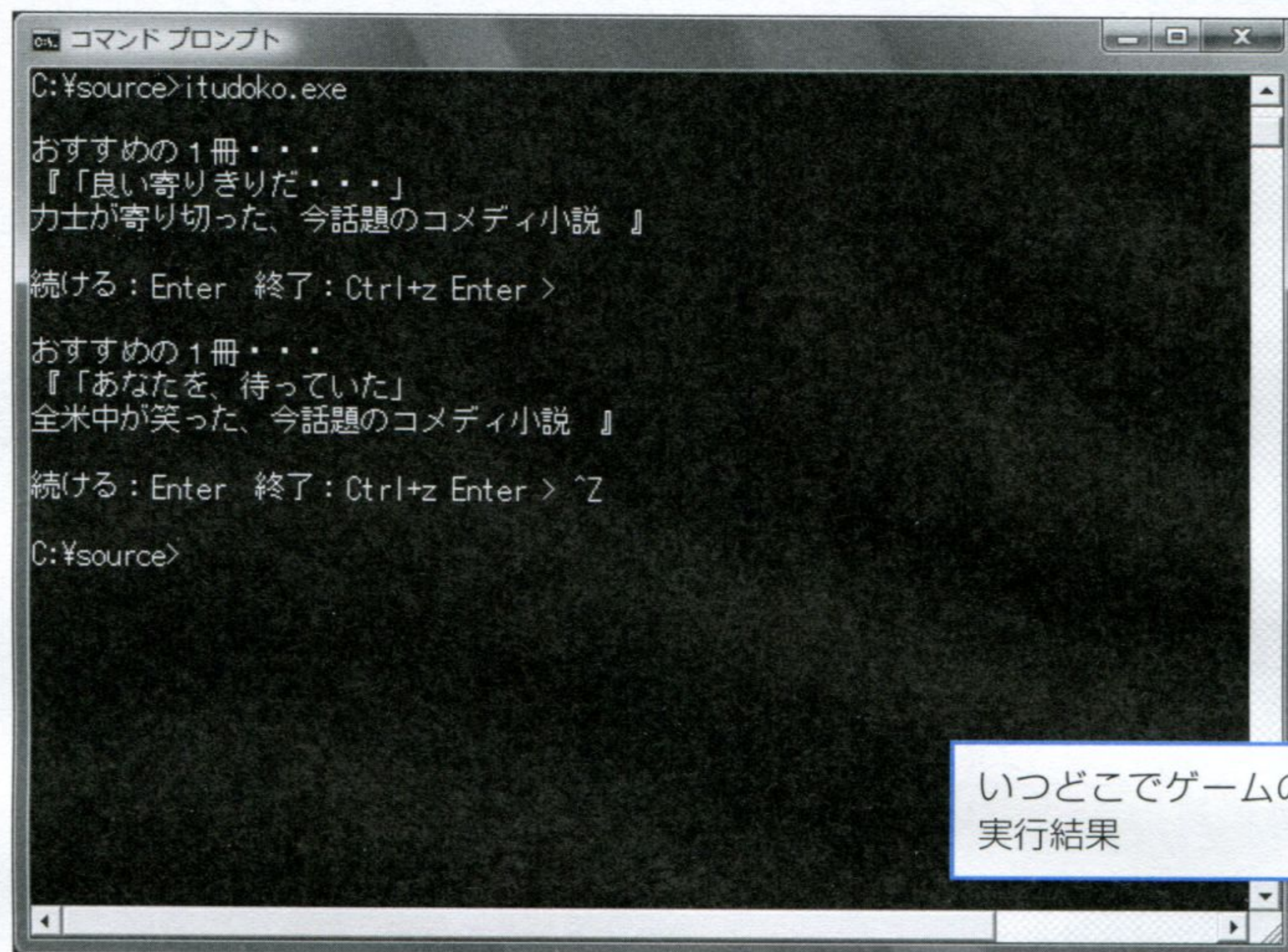
```
fseek( ファイルポインタ , オフセット , 指定位置 );
```

第3引数に指定した記号から、オフセット分ずらした位置に移動します。指定位置の記号は次のとおりです。オフセットとは、基準となる位置から目的の地点までの距離を表した値です。

●fseek関数の第3引数の指定値

指定位置	意味
SEEK_CUR	現在位置
SEEK_SET	ファイルの先頭
SEEK_END	ファイルの末尾

次の例では、最初はファイル末尾から10バイト分前の位置に移動し、次はファイルの先頭から5バイト目の位置に移動します。

```
fseek(fp, -10, SEEK_END);
fseek(fp, 5, SEEK_SET)
```

まとめ

fread関数、fwrite関数については今回のプログラムで使用しませんが、大事な関数なので、ここでしっかり理解しておきましょう。

第6日 3時限目

いつどこでゲームを完成させよう

【いつどこでゲームを作ろう③】

いつどこでゲームプログラムを完成させます。
結果の表示は、繰り返し何度もできるようにしましょう。

今回作成する例題

```
C:¥source>itudoko.exe

おすすめの１冊・・・
『「良い寄りきりだ・・・」
力士が寄り切った、今話題のコメディ小説　』

続ける：Enter　終了：Ctrl+z Enter >

おすすめの１冊・・・
『「あなたを、待っていた」
全米中が笑った、今話題のコメディ小説　』

続ける：Enter　終了：Ctrl+z Enter > ^Z

C:¥source>
```

いつどこでゲームの
実行結果

サンプルファイルはこちら　10days_c ▶ day06-03 ▶ itudoko.c

●このレッスンのねらい

いつどこでゲームのデータファイルからデータを読み込んで表示します。
データは区分ごとに分かれているので、それを区分ごとのポインタ配列に格納しますが、このとき、データを格納するために「メモリの確保」を行わなければなりません。この時限では、「メモリの確保」を学習しつつ、いつどこでゲームプログラムを完成させましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

int main() {
     FILE *fp; // 入力ファイルのファイルポインタ
     char datafile[] = "itudoko.txt";
     int data_max = 100;   // 各データ数
     char *serifu[data_max];     // セリフデータ
     char *hito[data_max]; // 人データ
     char *shita[data_max];      // 行動データ
     char *shurui[data_max];     // 種類データ
     char kubun[8], naiyou[256];
     int serifu_c= 0, hito_c = 0, shita_c = 0, shurui_c = 0;// 各データの数
     char buff[512]; // 読み込んだ文字列を格納する
     int i;
     int c;

     // 入力ファイルをオープンする
     if((fp = fopen(datafile, "r")) == NULL) {
          printf(" ファイルオープンエラー¥n");
          exit(1);
     }
     while(fgets(buff, sizeof buff, fp) != NULL){
          sscanf(buff, "%s¥t%s", kubun, naiyou);
          if((strcmp(kubun, "serifu") == 0) && (serifu_c < data_max)) {
               serifu[serifu_c] = (char*)malloc(strlen(naiyou) + 1);
               strcpy(serifu[serifu_c++], naiyou);
          } else if((strcmp(kubun, "hito") == 0) && (hito_c < data_max)) {
               hito[hito_c] = (char*)malloc(strlen(naiyou) + 1);
               strcpy(hito[hito_c++], naiyou);
          } else if((strcmp(kubun, "shita") == 0) && (shita_c < data_max)) {
               shita[shita_c] = (char*)malloc(strlen(naiyou) + 1);
               strcpy(shita[shita_c++], naiyou);
          } else if((strcmp(kubun, "shurui") == 0) && (shurui_c < data_max)) {
               shurui[shurui_c] = (char*)malloc(strlen(naiyou) + 1);
               strcpy(shurui[shurui_c++], naiyou);
```

```
        }
    }
    fclose(fp);

    srand(time(NULL));
    do {
        printf("¥nおすすめの1冊・・・¥n");
        printf("『「%s」¥n%sが%s、今話題の%s小説 』¥n¥n",
            serifu[rand()%serifu_c], hito[rand()%hito_c],
            shita[rand()%shita_c], shurui[rand()%shurui_c]);
        printf("続ける：Enter  終了：Ctrl+z Enter > ");
    } while((c = getchar()) != EOF);
    for(i = 0; i < serifu_c; i++) { free(serifu[i]); }
    for(i = 0; i < hito_c; i++) { free(hito[i]); }
    for(i = 0; i < shita_c; i++) { free(shita[i]); }
    for(i = 0; i < shurui_c; i++) { free(shurui[i]); }
    return 0;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「itudoko.c」という名前*1で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、itudoko.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o itudoko itudoko.c
```

4 プログラムを実行する

```
C:¥source>itudoko.exe
```

▼

```
おすすめの1冊・・・
『「あなたを、待っていた」
おすもうさんが涙した、今話題の携帯小説』
続ける：Enter  終了：Ctrl+z Enter > ^Z
```

データファイルのデータをランダムに組みあわせた文が表示されれば成功！

解説

1 メモリの確保と開放

プログラムを実行すると、プログラムの中で使う変数のデータはメモリ上に確保され、プログラム終了時に開放されます。

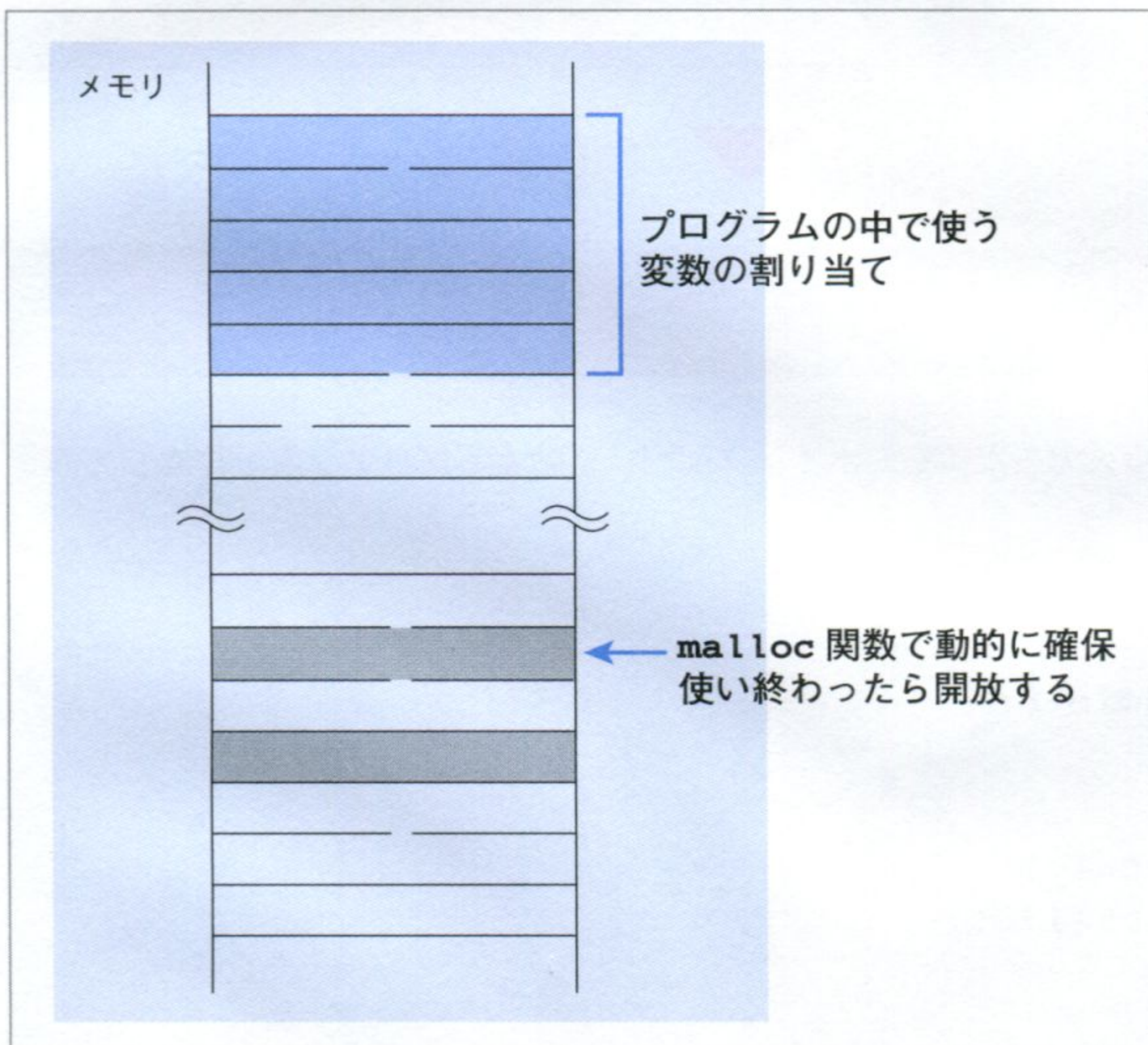
C言語では、プログラム中で定義されている変数以外にもプログラム実行中にメモリを動的に確保して、使いおわったら開放することができます。

メモリの確保にはmalloc関数を使います。

【malloc関数】

```
malloc( 確保するメモリのバイト数 )
```

malloc関数の引数に確保するメモリの大きさを指定すると、確保した領域の先頭アドレスが返されます。メモリが不足している場合は領域が確保できないので、NULLポインタが返されます。

●メモリの確保とmalloc関数

確保した領域は、必要がなくなったらfree関数を使って開放します。

【free関数】

```
free( 確保したメモリのアドレス )
```

malloc関数、free関数ともに、使用するにはstdlib.hをインクルードします。

128バイトのメモリを確保して開放するプログラムを作成してみましょう。

【6-3_sample1.c】

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *data;

    data = (char*)malloc(128);
    if (data == NULL) {
        printf("メモリ確保に失敗しました¥n");
        exit(1);
    }
    printf("メモリを確保しました¥n");
    free(data);
    return 0;
}
```

```
C:¥source>6-3_sample1.exe
メモリを確保しました
```

次に、標準入力から入力した値を、ポインタ配列に格納するプログラムを作成してみます。

【6-3_sample2.c】

```
#include <stdio.h>

int main() {
    char *data[3];
    char input[128];
    int i;

    for(i = 0; i < 3; i++) {
        printf("入力してください > ");
        gets(input);
        data[i] = input;
    }
    for(i = 0; i < 3; i++) {
        printf("%d番目の入力値：%s¥n", i+1, data[i]);
```

```
    }
    return 0;
}
```

これを実行すると、ポインタ配列dataの内容はどれも3番目の入力値になってしまいます。

```
入力してください > あいうえお
入力してください > かきくけこ
入力してください > さしすせそ
1番目の入力値：さしすせそ
2番目の入力値：さしすせそ
3番目の入力値：さしすせそ
```

ポインタ配列の各要素は、次のようにイコールで代入することができました。

```
data[0] = "あいうえお";
data[1] = "かきくけこ";
data[2] = "さしすせそ";
```

先ほど作成した6-3_sample2.cの入力部分を、この3つに置き換えてみます。すると、data[0]～data[2]までの出力はそれぞれ上記の代入値どおりになります。

この場合、プログラムの実行時に文字列「あいうえお」分の領域がメモリに確保され、その先頭アドレスがdata[0]に代入されたのです。同様に、data[1]には「かきくけこ」の先頭アドレスが代入されます。

修正前の6-3_sample2.cでは、data[i]にすべてinput配列の先頭アドレスが入ってしまったので、data[0]～data[2]の値はすべて同じinputを指しています。このため、あとから参照すると、全て同じ最後に入力された3番目の値になってしまいます*2。

では、正確に入力値をポインタ配列に格納するには、どうしたらよいでしょう？

こんなときこそ、動的メモリ確保関数であるmalloc関数の出番です。次のプログラムを見てください。

ヒント

*2：両方の場合において、全データのアドレスを表示してみるとわかりやすいと思います。

printf("%p %p %p¥n", &data[i], *(data+i), &input);

前から、data[i]のアドレス、data[i]の値が指し示しているアドレス、inputのアドレスです。

【6-3_sample3.c】

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
    char *data[3];
    char input[128];
    int i;

    for(i = 0; i < 3; i++) {
        printf("入力してください > ");
        gets(input);
        data[i] = (char*)malloc(strlen(input) + 1);*3
        strcpy(data[i], input);
    }
    for(i = 0; i < 3; i++) {
        printf("%d番目の入力値：%s¥n", i+1, data[i]);
    }

    for(i = 0; i < 3; i++) { free(data[i]); }
    return 0;
}
```

ヒント

*3：以降、メモリ確保失敗時の処理は省略します。

まず、入力値の長さ（+1）の大きさをmalloc関数で確保し、その先頭アドレスをdata[i]に代入します。領域が確保されたので、そのときの入力値であるinputの値をdata[i]にコピーします。

次の入力ではまた同じことを繰り返します。新しく入力された分の文字列の大きさをメモリ上に確保し、data[i]がそこを指すようにします。

このように入力ごとに文字列のメモリ確保を行い、dataの各要素はそれを指すようにすれば、入力値の保存ができます。

結果は次のようになります。

```
入力してください > test
入力してください > あいうえお
入力してください > 123
1番目の入力値：test
2番目の入力値：あいうえお
3番目の入力値：123
```

各入力値が正確に保存されています。
最後に、使いおわった領域をfree関数を使って開放しましょう[*4]。

> **ヒント**
> *4：free関数を使わなくてもプログラム終了と同時にメモリは開放されますが、環境により不具合が出ることがあるので、なるべくちゃんと呼び出しましょう。

2 いつどこでデータファイルの読み込み

いつどこでゲームのデータファイルは、1行1データで「区分」とその「内容」がタブ区切りになっています。1時限目にデータの読み込みを行うプログラム（6-1.c）を作りましたが、6-1.cでは読み込んだデータを表示しただけで、プログラムの中でデータを保存していません。

いつどこでゲームでは、データファイルから読み込んだデータを組みあわせて文章を作るので、先にデータファイルからのデータを読み込んで保存しておく必要があります。

データは4つの区分に分かれるので、ポインタ配列を4つ用意しておきます。

```
int data_max = 100; // 各データ数
char *serifu[data_max]; // セリフデータ
char *hito[data_max];    // 人データ
char *shita[data_max];   // 行動データ
char *shurui[data_max]; // 種類データ
int serifu_c= 0, hito_c = 0, shita_c = 0, shurui_c = 0;// 各データの数
```

読み込んだデータを変数kubunとnaiyouに分け、malloc関数を使ってメモリを確保してから、区分別のポインタ配列にデータ内容を格納します。

```
while(fgets(buff, sizeof buff, fp) != NULL){
    sscanf(buff, "%s¥t%s¥n", kubun, naiyou);
    if((strcmp(kubun, "serifu") == 0) && (serifu_c < data_max)) {
        serifu[serifu_c] = (char*)malloc(strlen(naiyou) + 1);
        strcpy(serifu[serifu_c++], naiyou);
    } else if((strcmp(kubun, "hito") == 0) && (hito_c < data_max)) {
        (略)
    }
}
```

セリフのデータを読み込んだら、セリフデータの数である変数serifu_cをインクリメントします。他の区分データもすべて同じ処理を行います。すると、最終的にデータファイルの全データは、4つのポインタ配列に分類、格納されます[*5]。

> **ヒント**
> *5：データは区分ごとに最高100個までを上限とし、それより多くなった場合は切り捨てます。

3 いつどこでゲームの実行

いつどこでゲームを実行してみましょう。

ポインタ配列に格納されたデータを使って文を作ります。区分データからランダムに選んで表示しましょう。

```
printf("『「%s」¥n%s が %s、今話題の %s 小説　』",
     serifu[rand()%serifu_c], hito[rand()%hito_c],
     shita[rand()%shita_c], shurui[rand()%shurui_c]);
```

表示は繰り返し行えるように、do～while文を使います。表示を続ける場合はただ［Enter］キーを押し、終了する場合は［Ctrl］＋［Z］*6キーを押します。標準入力で［Ctrl］＋［Z］はEOFになるので、それを終了条件とします。

ゲームがおわったら確保したメモリを開放して、終了します。

ヒント

*6：[Ctrl] ＋ [Z] は、UNIX環境では [Ctrl] ＋ [D] になります。

```
for(i = 0; i < serifu_c; i++) { free(serifu[i]); }
```

まとめ

いつどこでゲームを完成させるついでに、メモリ確保関数であるmalloc関数について学習しました。

もしもmalloc関数を使わない場合は、あらかじめ適当な大きさの領域を確保しておかなければならないので、最終的に使用しなかった分の領域は無駄になります。malloc関数は非常に便利な関数ですが、使い方を間違うと逆にメモリの無駄を引き起こしてしまうこともあるので、注意して使いましょう。

練習問題

Q　2時限目に作成したitudoko_data.cを、区分ごとに同じ内容は追加できないよう、malloc関数を使って改良しなさい。

［ヒント］
ファイルのオープンモードは変更せず、ファイルを開いたら、最初に先頭にファイルポインタを移動して読み込みを行う。

解答は巻末に

COLUMN

文字型なのにintを指定する入力方法

itudoko.cでは、表示を続けるかどうかの入力をgetchar関数で取得しました。getchar関数は文字を読み込む関数ですが、読み込んだデータを格納する変数にはint型を使いました。今回のプログラムのように入力が終了したとき（[Ctrl] ＋ [Z]を入力したとき）に、getchar関数はEOFを読み込みます。このEOFは通常は「-1」で、char型の範囲外[*7]となるので、int型を使っています。
文字を数値型で受け取って問題はないのか？ と思う方もいるでしょう。
scanf関数でも同じことがいえるので、こちらを例に見てみましょう。

```
#include <stdio.h>

int main() {
    int c; // 入力文字を格納する変数

    scanf("%c", &c); // 入力文字を受け取る
    printf("%c", c); // 入力した文字を出力
    return 0;
}
```

```
a ←—— 入力文字
a ←—— 入力した文字を出力
```

変数の宣言はint型なのに、scanf関数での入力指定はchar型"%c"で値が取得できています。
文字型のデータは、数値としても扱うことができるのです。
コンピュータ上で扱うデータは、すべて1と0で表現されていることを、第2日の章末で説明しました。つまり、コンピュータ上で扱うデータは、0と1で成り立つ表現方法である、2進数に置き換えることができます。さらに2進数は10進数へ変換できます。
例えば、文字「a」は2進数で01100001と表されます。これを10進数に直すと97です。つまり、文字「a」は97という数値に対応づけられています[*8]。
「a」の他にも、半角英数文字や半角記号文字は、それぞれきめられた10進数の数値が存在します。この対応づけがなされている文字をASCII文字、また、数値との対応表をASCIIコード表といいます。
ASCII文字以外の半角カナや全角（2バイト）文字も、それぞれきめられた数値により表現されています。ただし、ASCII文字以外は表現する文字コード体系により、その数値は異なります。

次の出力を実行してみると、両方結果は「数値：97 文字：a」となります。

ヒント

*7：通常、char型は「-128～127」の範囲を扱いますが、処理系によっては0～255で文字コードを扱うので、この「-1」が扱えません。int型にしておけばどの環境でも-1を受け取ることができます。

ヒント

*8：文字「1」を数値に直すと1ではなく、ASCIIコード表示対応した数値49になります。

ヒント

*9：文字は数値としても扱うことができるので、計算することもできます。例えば、'a'+3の結果を数値で表すと、100になります。

```
printf("数値：%d 文字：%c ¥n", 'a', 'a');
printf("数値：%d 文字：%c ¥n", 97, 97);
```

文字は数値として扱うことができる[*9]ことをおぼえておきましょう。

COLUMN

用語の読み方

メモリを動的に確保する「malloc」関数、最初に見たときどのように読みましたか？ 例えば整数データ型のint、これはたいていの人がそのまま「イント」と読むでしょう。では、文字のデータ型を表すcharはどうでしょう。これは「キャラ」という人もいれば「チャー」を使う人もいます。もしかしたら「チャラ」の人もいるかもしれません。
これらプログラムで使用する用語の読み方は、絶対にこう読むべき！ というものはありません。好きなように読みましょう。

先ほどのmallocは、intと同じようにそのまま読めば「マロック」ですが、筆者は「エム・アロック」を使います。「マロック」でも「エム・アロック」でも、人に説明するときに「malloc」として通じればいいのです。わからなければ書いて説明すれば何の問題もありません。他にも、文字列を操作する関数の最初につく「str」(「strcopy」など）は「ストラ」だったり、丁寧に「ストリング」という人もいます。

会社や学校、その中でもさらに部署ごとや学科ごとに、独自の読み方をする場合があります。一種の方言です。筆者が聞いた中ではかなり傑作なものもありました。
読み方がわからない用語は、まわりの人にどう読むか聞いてみるのもよいでしょう。人それぞれで、結構面白いです。

第 7 日

ブラックジャックゲームを作ろう

1時限目　関数を利用しよう

2時限目　トランプのカードを表示しよう

3時限目　2次元配列を理解しよう

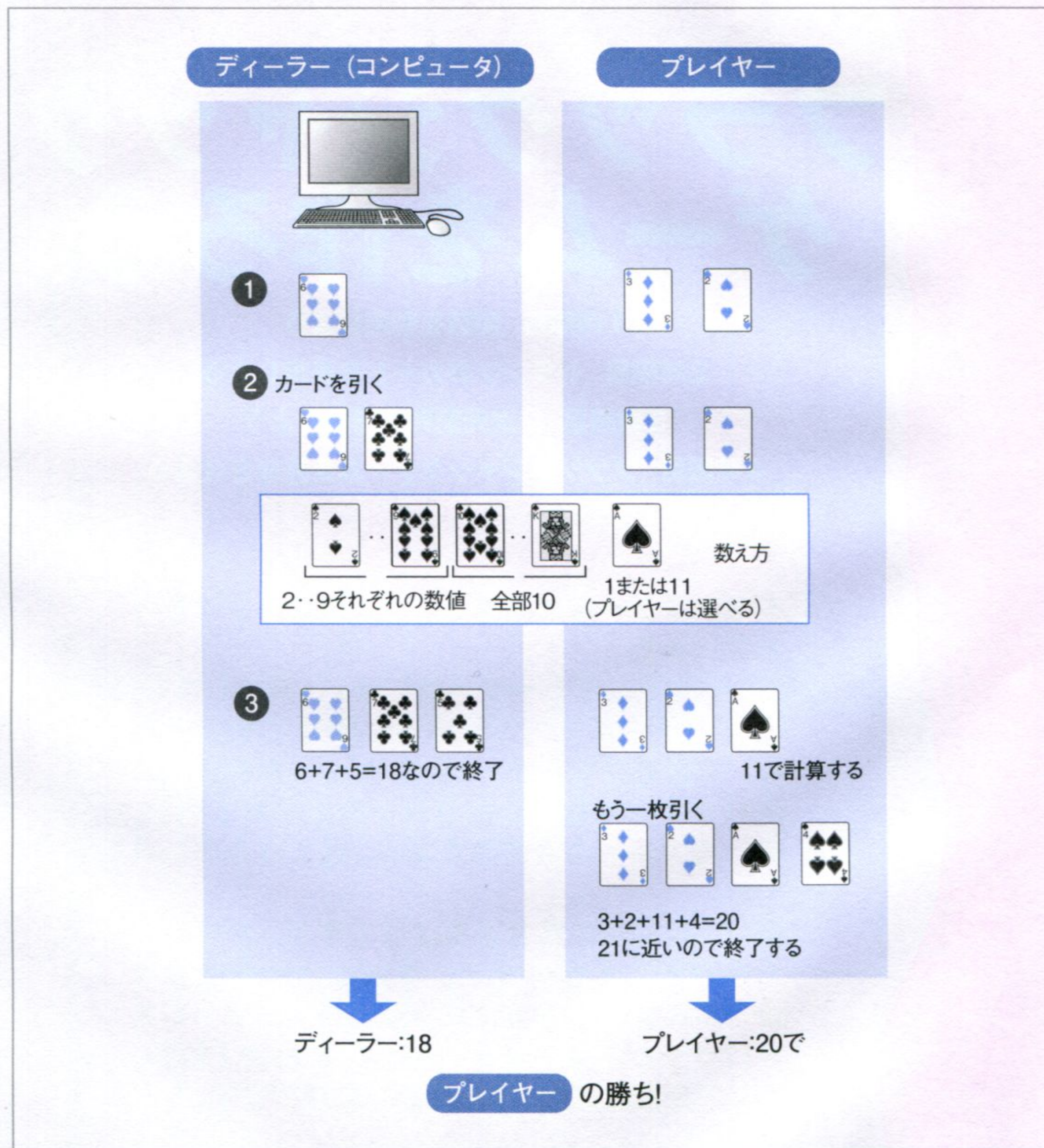
4時限目　ブラックジャックゲームを完成させよう

トランプゲームの定番、ブラックジャックを作りましょう。
プレイヤーとディーラーの1対1で勝負します。ディーラーはコンピュータにやってもらうので、今回もプレイヤー対コンピュータの勝負になります。お互いにトランプのカードを引き、引いたカードの合計数を21に近づけるゲームです。
遊び方はいろいろとあるようですが、プログラムを作ることを考えて、今回は比較的簡単なルールを採用します。ルールに沿ったゲームをプログラムで実現するために、今日は「関数」と「2次元配列」について新しく学びます。

今日作るプログラムについて

ブラックジャックゲーム

ブラックジャックゲームはディーラー（コンピュータ）とプレイヤーでカードを引き、引いたカードの合計数を21に近づけるゲームです。

ディーラー（コンピュータ）
プレイヤー
❶
❷ カードを引く
2‥9それぞれの数値
全部10
1または11
(プレイヤーは選べる)
数え方
❸
6+7+5=18なので終了
11で計算する
もう一枚引く
3+2+11+4=20
21に近いので終了する
ディーラー:18
プレイヤー:20で
プレイヤー の勝ち!

1 まず、ディーラーがトランプのカード2枚をプレーヤーに配ります。ディーラー自身にも1枚配ります。この3枚だけは表向きにするので、どのカードを引いたか表示します。

2 ここからが勝負です。ディーラーは自分のカードの数の合計数が16以下の場合は、カードを引き続けます。ディーラーの2枚目以降は、どのカードを引いたか表示しません。
カードの数え方は、2～9はそのままの数値として数え、絵札（キング、クイーン、ジャック）と10は、すべて10として数えます。
エースは、1または11として、状況に応じてプレーヤーの都合のよい方にカウントします。今回の場合、ディーラーはコンピュータが行うので、ディーラーがエースを引いたときは、すべて11としてカウントします。

3 プレイヤーは、カードの合計が21を超えるまでは、好きなだけカードを引くことができます。合計が21を超えるとプレイヤーの負けです。ディーラーの合計が21を超えていたらディーラーの負けです。
どちらも21を超えていなければ、21により近いほうが勝ちです。プレイヤーもディーラーも合計が21を超えていた場合か、両者同点の場合は、引き分けです。

つまり、プレイヤーは21を超えないように考えてカードを引けばいいのです。例えば、今のところの合計が15だった場合、ここで引くのをやめれば無難ですが、あと1枚引いて、そのカードが6以下ならば、勝てる可能性が高くなります。しかし7以上の数値が出てしまうと、負けになります。

ブラックジャックゲームの実際の動作

1 ブラックジャックゲームを実行すると、ディーラーの１枚目と、プレイヤーの引いたカードがすべて表示される

```
C:¥source>blackjack.exe
【ブラックジャック】
ディーラー一枚目：スペードの 4
他は伏せる

プレイヤー一枚目：スペードの J
プレイヤー二枚目：ダイヤの 10
もう 1 枚引きますか ?(y/n) >
```

プレイヤーがエースを引いたら、11で計算するかどうかを聞いてくるので、11で計算したい場合は「y」を入力し、1で計算したい場合は「n」を入力して、[Enter] キーを押す

```
プレイヤー一枚目：スペードのA
11として計算しますか？ (y/n) > y
```

2

「もう1枚引きますか？」に「y」を入力すると、カードを引き続ける

```
もう1枚引きますか？(y/n) > y
クローバーのQ
もう1枚引きますか？(y/n) >
```

3

カードを引くのをやめるときは、「n」を入力

```
もう1枚引きますか？(y/n) > n
```

4

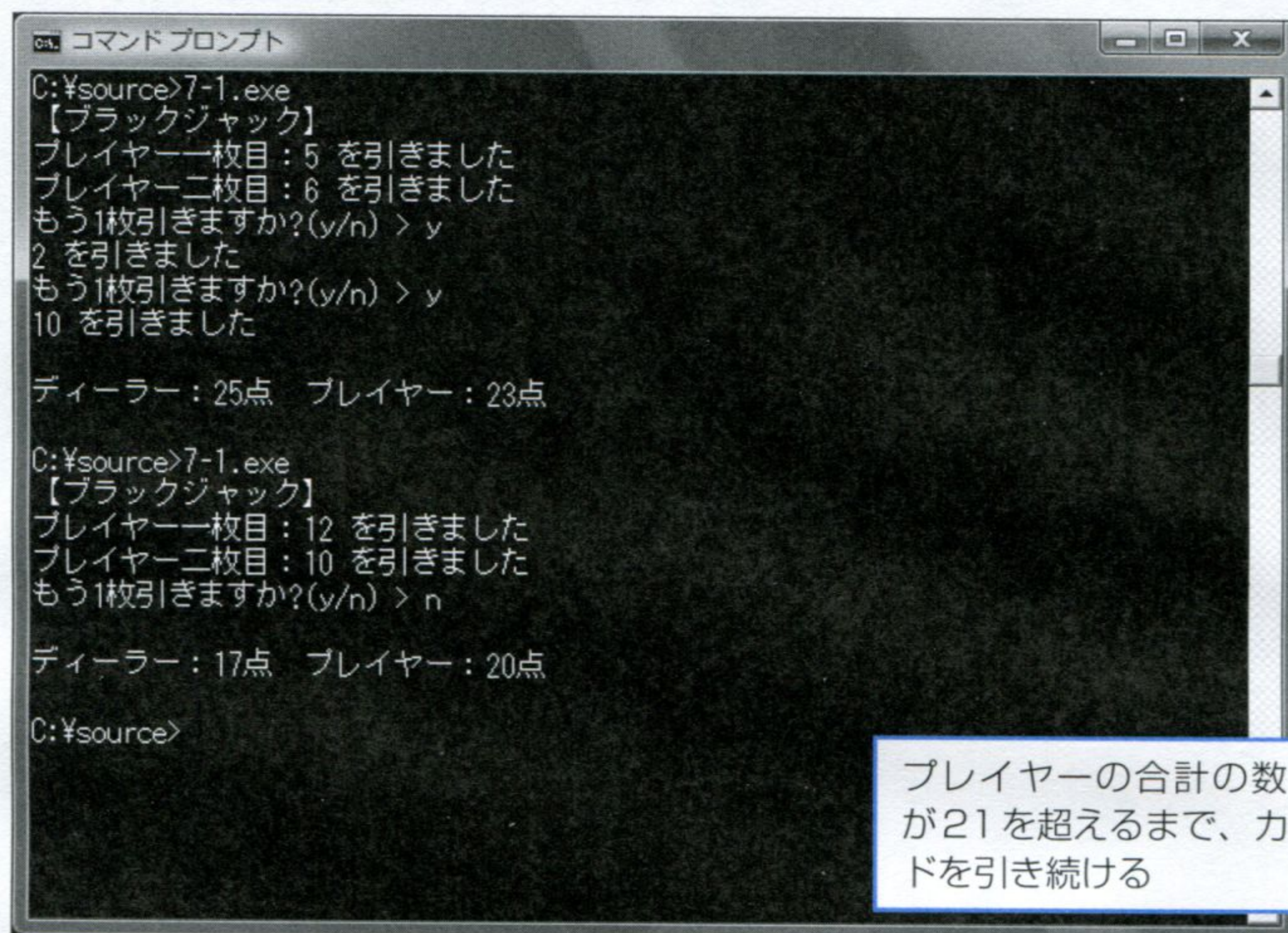
カードを引くのをやめるかプレイヤーの引いたカードが21を超えたら自動的に終了し、どちらが勝ったか判定して結果を表示する。判定結果は「プレイヤーの勝ち！」「ディーラーの勝ち！」「引き分け」の3種類

```
C:¥source>blackjack.exe
【ブラックジャック】
ディーラー一枚目：クローバーの9
他は伏せる

プレイヤー一枚目：クローバーの3
プレイヤー二枚目：ダイヤのA
11として計算しますか？ (y/n) > y
もう1枚引きますか？(y/n) > y
スペードの5
もう1枚引きますか？(y/n) > n

ディーラー：20点　プレイヤー：19点
ディーラーの勝ち！
```

```
C:¥source>blackjack.exe
【ブラックジャック】
ディーラー一枚目：ダイヤの６
他は伏せる

プレイヤー一枚目：スペードの３
プレイヤー二枚目：ハートの２
もう１枚引きますか？(y/n) > y
クローバーの４
もう１枚引きますか？(y/n) > y
スペードのQ
もう１枚引きますか？(y/n) > n

ディーラー：26点　プレイヤー：19点
プレイヤーの勝ち！
```

```
C:¥source>blackjack.exe
【ブラックジャック】
ディーラー一枚目：クローバーの３
他は伏せる

プレイヤー一枚目：ダイヤのK
プレイヤー二枚目：ダイヤの４
もう１枚引きますか？(y/n) > y
スペードのK

ディーラー：26点　プレイヤー：24点
引き分け
```

第7日 1時限目

関数を利用しよう【ブラックジャックゲームを作ろう①】

ディーラーはカードの合計が16以下ならカードを引き続け、プレイヤーは、3枚目以降は合計数値が21を超えるまで、好きなだけカードを引き続けるプログラムを作ります。

今回作成する例題

```
C:¥source>7-1.exe
【ブラックジャック】
プレイヤー一枚目：5 を引きました
プレイヤー二枚目：6 を引きました
もう1枚引きますか?(y/n) > y
2 を引きました
もう1枚引きますか?(y/n) > y
10 を引きました

ディーラー：25点　プレイヤー：23点

C:¥source>7-1.exe
【ブラックジャック】
プレイヤー一枚目：12 を引きました
プレイヤー二枚目：10 を引きました
もう1枚引きますか?(y/n) > n

ディーラー：17点　プレイヤー：20点

C:¥source>
```

プレイヤーの合計の数値が21を超えるまで、カードを引き続ける

サンプルファイルはこちら　10days_c ▶ day07-01 ▶ 7-1.c

●このレッスンのねらい

ブラックジャックでは、プレイヤーとディーラーがお互いに何回かカードを引きます。ルールどおりにプログラムを書くと、カードを引く機会はこれだけあります。

①ディーラーが1枚引く
②プレイヤーが2枚引く
③ディーラーは合計が16以下のときに引き続ける
④プレイヤーは21に近づくよう引き続ける

「カードを引く」行為はプレイヤーもディーラーも同じです。プログラムを書くと、「カードを引く」処理をルールの流れに沿って何度も書かなければなりません。特に「カードを引く」処理が複数行のプログラムになる場合、何度も同じ処理を書くのは、プログラムとしてあまり美しくありません。
これを解決するのが「関数」です。今まではC言語で用意されている関数のみを使ってきましたが、関数は自分で作ることもできます。「カードを引く」処理を書いた関数を作って用意しておき、それを呼び出せばプログラムがすっきりします。この時間は関数について学習しましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

// カードの点数
int total[13] = { 11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10
};

/* カードを引く関数
引数 h：カード内容表示フラグ
        プレイヤーターンの時は 1、ディーラーターンの時は 0
戻り値 r: 引いたカードの点数
*/
int drawCard (int h) {
    int draw_digit; // 引いたカードの数
    int r; // 引いたカードの点数

    draw_digit = rand() % 13 + 1;

    if(h) {
        printf("%d を引きました ¥n", draw_digit);
    }
    r = total[draw_digit-1];
    return r;
}

int main() {
    int dealer; // ディーラーの引いたカードの合計
    int player; // プレイヤーの引いたカードの合計

    char y_n; // カードを引くか引かないかの答え

    srand(time(NULL));
    printf("【ブラックジャック】¥n");

    // ディーラーの一枚目は今回省略

    // プレイヤー
```

```
    printf("プレイヤー一枚目：");
    player = drawCard(1);
    printf("プレイヤー二枚目：");
    player += drawCard(1);

    //ディーラーが引く
    do{
        dealer += drawCard(0);
    } while(dealer <= 16);

    //プレイヤーが引く
    while(player < 21) {
        printf("もう1枚引きますか?(y/n) > ");
        scanf("%c" , &y_n);
        while (getchar() != '¥n') {  }
        if(y_n == 'y') {
            player += drawCard(1);
        } else if (y_n == 'n') { break; }
    }

    printf("¥nディーラー：%d点　プレイヤー：%d点¥n", dealer, player);

    return 0;
}
```

ヒント

＊1：拡張子に注意して保存しましょう。

2 **入力できたら、「7-1.c」という名前[＊1]で、「C:¥source」ディレクトリ下に保存する**

3 **コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、7-1.cをコンパイルする**

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 7-1 7-1.c
```

4 **プログラムを実行する。プレイヤーの合計の数値が21を超えるまで引き続けることができれば成功！**

【プレイヤーの１、２枚目の合計が21を超えた場合】

```
C:¥source>7-1.exe
```

```
【ブラックジャック】
プレイヤー一枚目：10 を引きました
プレイヤー二枚目：1 を引きました

ディーラー：19点 プレイヤー：21点
```

【プレイヤーの１、２枚目の合計が21を超えない場合】

```
C:¥source>7-1.exe
```

```
【ブラックジャック】
プレイヤー一枚目：2 を引きました
プレイヤー二枚目：11 を引きました
もう１枚引きますか？(y/n) > y
5 を引きました
もう１枚引きますか？(y/n) > n

ディーラー：17点 プレイヤー：17点
```

解説

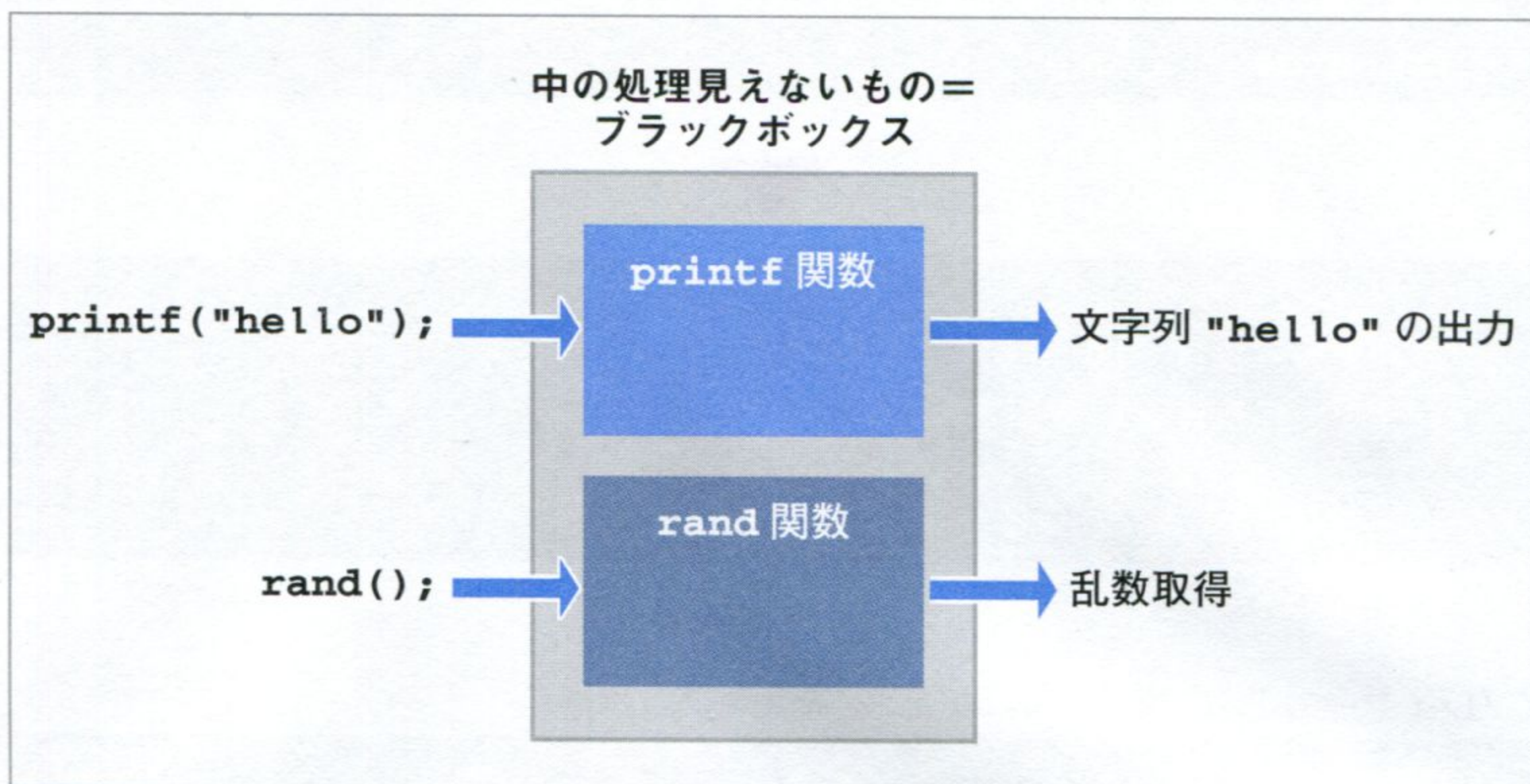
1 関数の基本

出力関数のprintf関数は、今までにも何度か出てきました。この関数の中でどういう処理を行っているのかはわかりませんが、とにかく書式どおりにprintf関数を使うと、指定した文字列が出力されました。rand関数も呼び出すだけでランダム数値を取得できました。

中で何をやっているかわからないものをブラックボックスといいます。

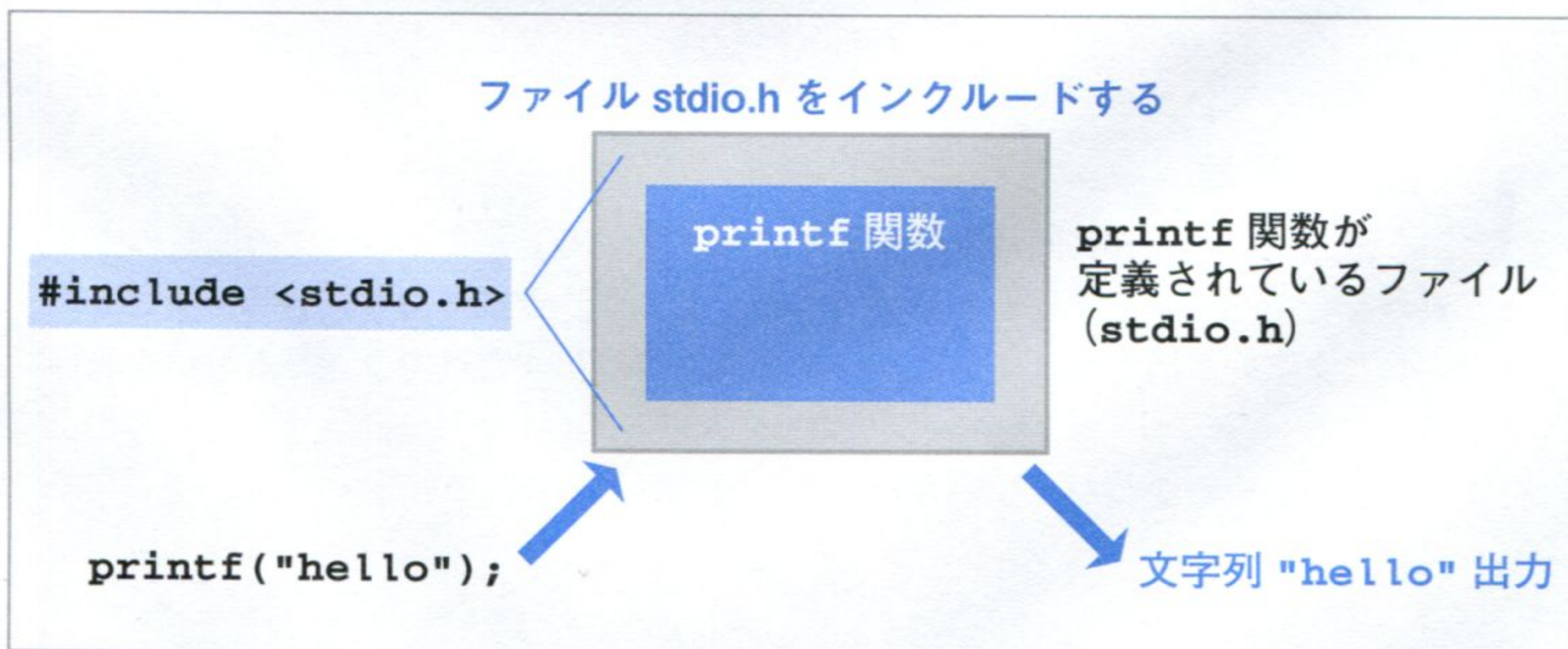
●ブラックボックス

中の処理見えないもの＝
ブラックボックス

入力	関数	結果
printf("hello");	printf 関数	文字列 "hello" の出力
rand();	rand 関数	乱数取得

しかし、中身を見ることができないだけで、実際はprintf関数の中で文字列を出力するための処理が行われています。

関数とは、「一連の処理をひとまとめにしたもの」です。関数は引数を持つことができ、与えた引数によって異なる結果を取得できます。printf("Hello!")では文字列「Hello!」が、printf("World")では文字列「World」が表示されます。

printf関数の一連の処理は、printf関数を使うときにプログラムの最初にインクルードするstdio.hファイルに書いてあります[*2]。

printf関数の中身の処理を書いた部分がどこかにあり、プログラムの最初にそれをインクルードして取り込んでいるので、プログラムの中でその関数を呼び出して使うことができます。この「中身の処理を書いた部分」を「関数定義」といいます。

ヒント
*2：正確には、stdio.hファイル自体に処理が書いてあるわけではありませんが、意味としては同じです。このあたりの詳細は第10日に学習します。

●関数定義

ファイル stdio.h をインクルードする

#include <stdio.h>

printf 関数

printf 関数が
定義されているファイル
(stdio.h)

printf("hello");

文字列 "hello" 出力

2 自作関数の書き方のきまり

C言語であらかじめ用意されているものとは別に、自分で関数を作ることもできます。ある特定の部分の処理をまとめて関数を作りましょう。ここでは、何度も出てくる「カードを引く」処理を書いた部分を関数にして、プログラムをすっきりさせます。

自分で作る関数には書き方のきまりがあります。それに従って関数を作ります。

(1) 関数名をきめる

関数には名前（関数名）をつけます。これは変数名と同じで半角英数文字を使って自由につけます。例えば、「カードを引く」関数ならば「drawCard」などとつけておくと、何をする関数なのかがすぐわかります。ですが、今はとりあえずfunc1という名前にしておきます。

(2) 関数の型をきめる

関数には「戻り値」があります。戻り値[*3]とは、関数を呼び出したときに返ってくる値のことです。例えば、乱数を求めるrand関数の場合、

```
x = rand();
```

とすると、変数xにランダムな値が入りました。rand関数を呼び出すと、整数型のランダムな値が返ってきます、これが関数の戻り値になります。

そして、返ってくる値のデータ型が関数の型になります。rand関数は整数値を返すので、関数の型はint型です。

rand関数は、ランダムな数値を作り出す処理を行い、最後にその値を返すよう定義されています。rand関数を使う側は、その戻り値を受け取るだけでいいのです。

```
x    =    rand();
```

戻り値は int 型＝ rand 関数は int 型

自分で作る関数にも型が必要になります。自作関数func1が整数値を返すならば、その自作関数の型はint型です。実数を返すならdouble型で定義をします。しかし、ある一定の処理だけを行い、何も値を返さない関数もあります。その場合の型はvoid型になります。

(3) 関数の引数をきめる

printf関数は、出力したい文字列を引数に指定します。自分で作る関数にも、何か引数を渡すことができます。rand関数のように引数を指定しない関数も作れます。

では、この3つのきまりに従って、func1関数の定義を書いてみましょう。

3 自作関数を定義する

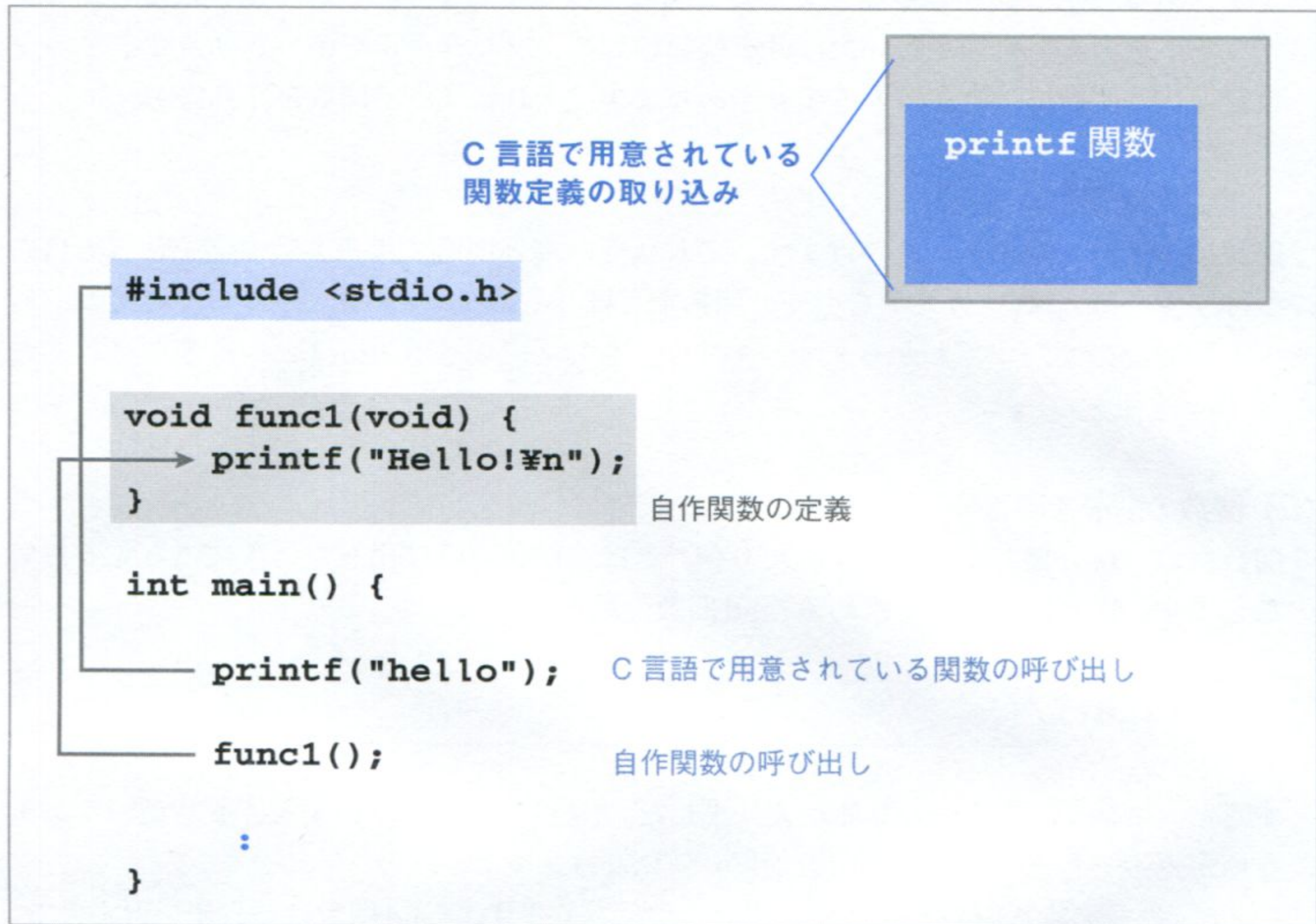
C言語であらかじめ用意されている関数の定義は、他のファイルで行っています。プログラムではそれをインクルードします。

ヒント

*3：戻り値は、「返り値」や「リターン値」とも呼ばれています。

一方、自分で作る関数の定義は、プログラムの中に書きます。

●自作関数の定義イメージ


```
#include <stdio.h>

void func1(void) {
    printf("Hello!¥n");
}

int main() {

    printf("hello");

    func1();

        :
}
```


関数定義は次のように書きます。

【関数定義】

```
関数の型 関数名 ( 引数 ) {
    処理 ;
}
```

(1) 引数と戻り値のない関数

まず、何も値を返さず、引数も使用しない関数を定義してみます。

```
void func1(void) {
    printf("Hello!¥n");
}
```

戻り値がないので、関数の型はvoid型です。引数が何もない場合、関数名のあとの ( ) の中にvoidと書いておきます。この関数を呼び出すには次のようにします。

```
func1();
```

では自作関数func1を定義し、main関数から呼び出すプログラムを作成します。次のように、関数の定義はmain関数の前、include文のあとに書くのが基本の形です。

【7-1_sample1.c】

```
#include <stdio.h>

// 自作関数 func1 の定義
void func1(void) {
    printf("Hello!¥n");
}

int main() {*4
    func1(); // 自作関数 func1 の呼び出し
    return 0;
}
```

```
Hello!
```

ヒント

*4：関数定義の引数voidは省略することもできます。
main関数は、プログラム実行時の引数を受け取らない場合は引数なしなのでmain()となりますが、main(void)が正式な書き方です。

(2) 戻り値のある関数

今度は戻り値のある関数を書いてみます。整数値を戻り値にする関数の場合、関数の型はint型です。このデータ型の値を、関数の中で行う処理の最後に、次のように書いて値を返します。

```
return 値;
```

では、整数値を戻り値として返すだけの関数を書いてみます。

【7-1_sample2.c】

```
#include <stdio.h>

int func2(void) { //int 型関数の定義
    return 10;
}

int main() {
    int d;
    d = func2();
    printf("%d¥n", d);
    return 0;
}
```

```
10
```

自作関数func2では、値10を返しています。整数値を戻り値としているので、関数func2はint型です。その結果を受け取り、main関数*5の中で出力しています。

ヒント

*5：main関数は、return 0;と最後に0を返しているので、int型です。

(3) 引数のある関数

次に、引数を使った関数を紹介します。ひとつの整数値を引数にする場合、その引数のデータの型（この場合はint型）と変数名を、関数名定義のあとの () の中に書きます。

```
void func3 （データ型 変数名） {
```

実際には次のようになります。

```
void func3 (int d) {
```

自作関数func3の処理ブロックの中では、この変数dを使うことができます。

main関数の中で自作関数func3を呼び出すときに引数に数値5を指定すると、func3の処理では変数dに値5が入ります。

●引数の値の渡され方

```
#include <stdio.h>

void func3(int d) {
    printf("%d¥n", d);
}

int main() {
    func3(5);
      :
}
```

変数dには
値5が渡される

では、引数に指定した値をそのまま返すだけの関数を作ります。

【7-1_sample3.c】

```
#include <stdio.h>

int func3(int d) {        //dは仮引数 *6
    return d;
}

int main() {
```

```
    int d;
    d = func3(1);
    printf("%d¥n", d);   //dは実引数*6
    return 0;
}
```

```
1
```

自作関数func3に引数1を渡し、その値をそのまま戻り値として返してもらっています。引数を2つ以上定義する場合、各引数はカンマ「,」で区切って、変数名とそのデータ型をセットで定義します。関数を呼び出す方も、渡す各引数をカンマで区切ります。

また、複数ある引数の型は、すべて同じである必要はありません。

```
func4(int d, char c) {
```

と定義すれば、第1引数を整数値、第2引数を文字にした関数ができあがります。この関数の呼び出し方は次のとおりです。

```
func4(5, 'a');
```

4 変数の範囲

ひとつの整数を引数として渡し、その引数値の2乗を計算して表示する関数を作ります。

【7-1_sample4.c】

```
#include <stdio.h>

void func5(int d) {
    int d2 = d * d;*7
    printf("%d¥n", d2);
}

int main() {
    func5(3);
    return 0;
}
```

ヒント

*6：関数定義の引数を仮引数、関数を使用するときに関数に引き渡す引数を実引数と呼びます。

ヒント

*7：自作関数の中で使う変数宣言も、その関数の中で最初にまとめて書きます。

9

引数3を2乗しているので、自作関数func5の中では値9を出力しています。

2乗した結果は自作関数func5の中の変数d2に代入されました。この変数d2をmain関数で参照することはできません。変数には、その値を参照したり代入したりする有効範囲が存在するからです。

(1) ローカル変数

例えば次のサンプルのように、その関数の中だけで使える変数をローカル変数といいます。

【ローカル変数の定義】

```
void func1(void) {
     int d = 2; // 関数 func1 のローカル変数
          (略)
}

int main() {
     char c; //main 関数のローカル変数
          (略)
}
```

自作関数func1の中の変数dは、自作関数func1の中だけで有効です。main関数の中に定義されている変数cは、main関数の中だけで有効です[*8]。自作関数func1の中でmain関数のローカル変数cを参照しようとすると、エラーになります。変数cは自作関数func1の中には存在しないからです。

ヒント

*8：自作関数func1の中でmain関数にあるのと同じ名前の変数cを作成した場合は、それはmain関数の変数cとは別のものになります。

●ローカル変数の有効範囲

```
#include <stdio.h>

void func1(void) {
     int d = 2; // 関数 func1 のローカル変数
                       参照できる
     print ("%d", d);
     print ("%c", c);         変数 d の有効範囲
}
                    参照できない！
int main() {
     char c; //main 関数のローカル変数
         :                    変数 c の有効範囲
}
```

(2) グローバル変数

自作関数でもmain関数でも使いたい共通の変数を作りたい場合は、関数宣言の外に変数を宣言します。変数宣言はインクルード文のあと、関数定義の前に書きます。

このような変数をグローバル変数と呼びます。有効範囲はプログラム全体です。

【7-1_sample5.c】

```
#include <stdio.h>

int d = 4; // グローバル変数

void func6(void) {
    printf("func6 %d¥n", d);
}

int main() {
    func6();
    printf("main %d¥n", d);
    return 0;
}
```

```
func6 4
main 4
```

どちらの関数の中でも、グローバル変数dの値が参照できました。

●グローバル変数の有効範囲

```
#include <stdio.h>

int d = 4; // グローバル変数

void func6(void) {
    printf("func6 %d¥n", d);
}

int main() {
    func6();
    printf("main %d¥n", d);
    return 0;
}
```

なお、グローバル変数で定義した変数名と同じ変数名をローカル変数として宣言した場合、ローカル変数が優先されます。

5 関数を利用したプログラム

では、本日の例題であるブラックジャックゲームについて説明していきます。

今までのプログラムと比べ、ブラックジャックゲームのプログラムは少し長くなります。長いプログラムを書くには、まず全体の骨組みをざっと作り、そこから中身をつけ足してプログラムを完成させるのがコツです。

この時限では、まずディーラーとプレイヤーがルールどおりにカードを引き、最後にお互いの合計点数だけを表示するところまで作ります。

(1) main関数の概要を考える

まずはディーラーがカードを引きます。本当なら1枚目は相手に見せる必要があるのですが、最初のプログラムではそれも省略します。

ディーラーの引いたカードの合計はint型の変数dealer、プレイヤーの引いたカードの合計はint型の変数playerとします。main関数の中で行う作業は次のとおりです。

●ブラックジャックゲームのmain関数での処理

```
int main() {
        int dealer; //ディーラーの引いたカードの合計
        int player; //プレイヤーの引いたカードの合計

        ディーラーが一枚カードを引く; ※(この時間のプログラムでは省略)

        プレイヤーが1枚目を引く;
        プレイヤーが2枚目を引く;

        dealerの値が16以下の間はカードを引き続ける処理;

        プレイヤーはplayerの値を21に近づくようカードを引き続ける;
                ※(条件:自分で引くのをやめるか、21を過ぎたら終了)

        dealerとplayerの値を表示する;
}
```

このように、ディーラーとプレイヤーがカードを引く処理が何度も出てきます。よって、この部分を関数化して、カードを引く自作関数drawCardを定義しましょう。

(2) カードを引く自作関数drawCardを考える

カードを引く自作関数drawCardについて考えてみましょう。

トランプのカードには4種類のマークがあり、それぞれ1〜13までの数字があります。しかし、ここではまだカードのマークについては考えません。1〜13までのどれかの数値

をrand関数[*9]を使って出せば、「カードを引く」ことになります。引いたカードの点数を関数の戻り値とします。この関数の型はint型になります。

ディーラーの引いたカードは、基本的にプレイヤーには見せませんが、プレイヤーは自分が引いたカードを知らなければ、ゲームを続けられません。よって、プレイヤーが引いた場合のみ、引いた数値を「○を引きました」と表示しましょう。ディーラーが引いた場合とプレイヤーが引いた場合を区別するには、関数の引数を利用します。

```
drawCard(0);
drawCard(1);
```

0を引数に関数を呼び出せばディーラーの番、1を引数に関数を呼び出したらプレイヤーの番とします。自作関数drawCardではこの引数を受け取って、今どちらが引いているのか判定し、プレイヤーのときだけ、引いた数を表示する処理を加えます。

最後にもう1点。引いたカードの値そのものが戻り値なら、1～13までの数値をそのまま返せばいいのですが、この関数では点数を返すので、1～13を対応する点数にそれぞれ変換する必要があります。そこで、1～13に対応する点数を格納した配列を用意します。

```
int total[13] = { 11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 };
```

エースの点数は、1か11かを選べますが、この配列では11と固定にしておきます。10以上はすべて10、あとはエース以外はカードの値がそのまま点数になります。配列の添え字は0からはじまるので、

```
total[引いた数-1]
```

が対応する点数になります。この点数配列は、グローバル変数にしておきます。

(3) カードを引き続ける

自作関数drawCardの処理がわかったところで、再びmain関数について考えてみます。

main関数の中で、プレイヤーとディーラーそれぞれがカードを引くときに、自作関数drawCardを呼び出します。1回カードを引くときは、自作関数drawCardを1回だけ呼び出します。(1) で説明した処理概要の「ディーラーの合計が16以下の間」と「プレイヤーの合計が21に近づくまで」の条件では、カードを何回引くのかはきまっていません。

ディーラーの場合は簡単です。16以下ならカードを引き続けるので、do～while文を使って、

```
do{
    dealer += drawCard(0);
} while(dealer <= 16);
```

とします。カードの合計が16を超えたら終了します。

ヒント

*9：rand関数で毎回違う値を出すために必要なsrand関数は、main関数の中で最初の方に1回だけ行います。自作関数drawCardの中で行うと、自作関数drawCardが呼び出されるたびに種を蒔いてしまうからです。

次にプレイヤーの場合を考えます。(1) で処理内容を見たように、繰り返しの前に2枚引いているので、その時点で合計が21以上になっている可能性もあります。よって、こちらはwhile文を使います。合計が21を超えていなければ、次の処理を繰り返します。

・カードを引くか引かないか聞く
・カードを引くを選択したら、カードを引き点数を加算する
・カードを引かないと選択したら、繰り返し処理を終了

カードを引くか引かないかは、文字「y」か「n」で答えるようにします。

```
char y_n; // カードを引くか引かないかの答え

while(player < 21) {
    printf("もう1枚引きますか?(y/n) > ");
    scanf("%c" , &y_n);
    while (getchar() != '¥n') {  }
    if(y_n == 'y') {
        player += drawCard(1);
    } else if (y_n == 'n') { break; }
}
```

入力値が「y」でも「n」でもなければ[*10]、何度でも繰り返しを行います。

ヒント

*10：予期せぬデータが入力された場合の処理は、自分で自由に設定してかまいません。例えば、入力値が「y」以外はすべて無条件で終了する、とプレイヤーに厳しく作ることもできます。

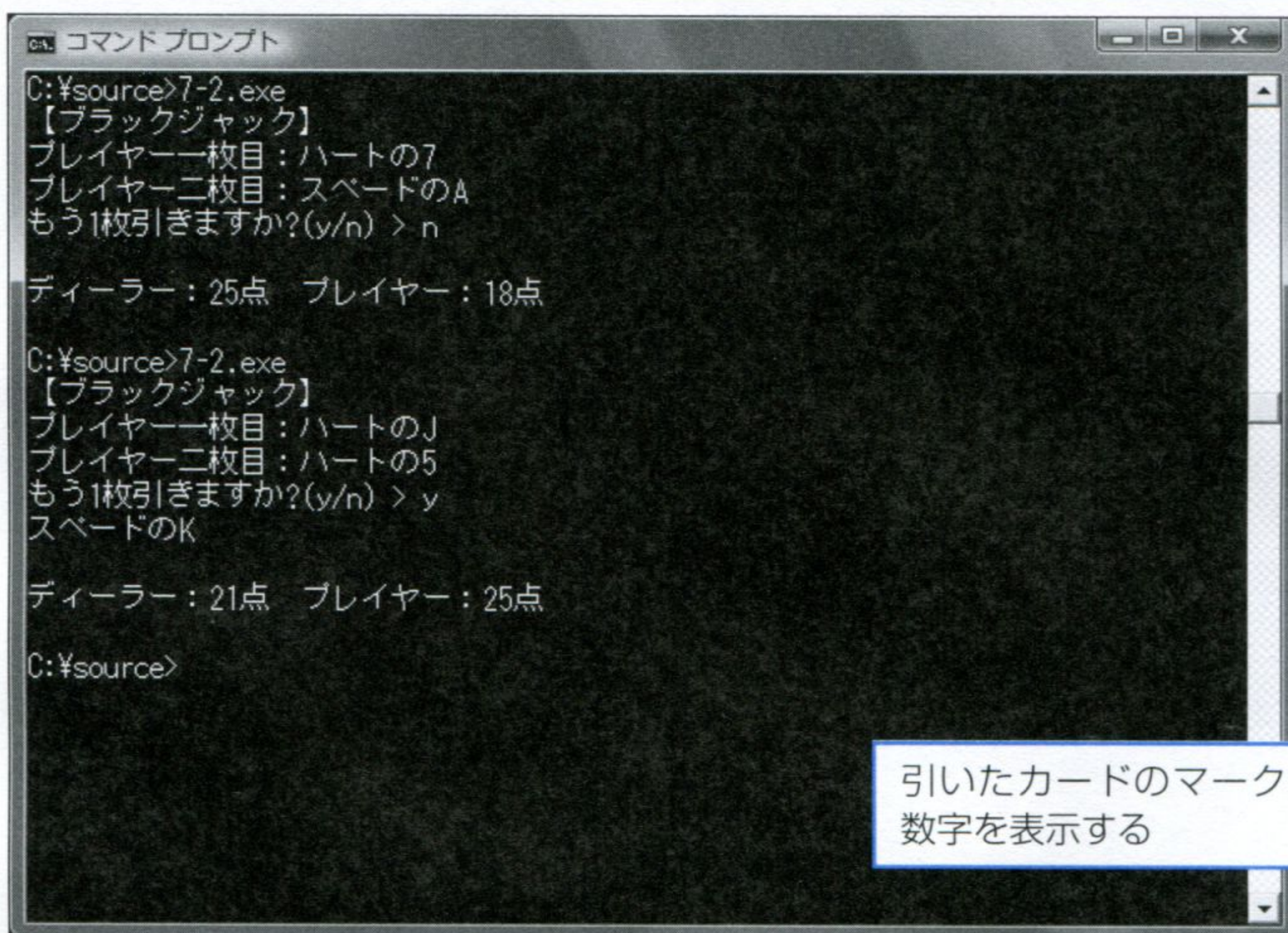
まとめ

この時限では、関数の基礎について学習しました。今までなんとなく使っていたものの意味がわかって、すっきりしたことでしょう。

今後作成するプログラムでは自作関数を作る機会が多くなるので、この時間に関数の基礎を身につけてしまいましょう。

練習問題

Q

プログラム実行時に２つの整数値を引数で渡して、その値を足した結果を返す関数を作りなさい。

[条件]
関数名や引数名は自由につけてよい

解答は巻末に

第7日 2時限目

トランプのカードを表示しよう

【ブラックジャックゲームを作ろう②】

1時限目のプログラムを応用して、引いたカードのマークも表示するプログラムを作ります。

今回作成する例題

```
C:¥source>7-2.exe
【ブラックジャック】
プレイヤー一枚目：ハートの7
プレイヤー二枚目：スペードのA
もう1枚引きますか?(y/n) > n

ディーラー：25点　プレイヤー：18点

C:¥source>7-2.exe
【ブラックジャック】
プレイヤー一枚目：ハートのJ
プレイヤー二枚目：ハートの5
もう1枚引きますか?(y/n) > y
スペードのK

ディーラー：21点　プレイヤー：25点

C:¥source>
```

引いたカードのマークと数字を表示する

サンプルファイルはこちら　10days_c ▶ day07-02 ▶ 7-2.c

●このレッスンのねらい

1時限目ではカードを引く部分だけを作りました。本レッスンでは引くカードの数だけでなく、どのマークの何のカード（数字）を引いたかを表示して、より「トランプのカードを引く」らしくしましょう。

今まではカードの数字も1から13までの数値だけでしたが、1だったら「A」、11だったら「J」とカードらしく表示します。

さらにプロトタイプ宣言を使って、自作関数をmain関数のうしろに書く方法も学習しましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

// カードの点数
char total[13] = { 11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 };
// カードのマーク
char *digit[13] = { "A", "2", "3", "4", "5", "6", "7",
                    "8", "9", "10", "J", "Q", "K" };
// カードの数値
char *mark[4] = {
    "ハート",
    "ダイヤ",
    "スペード",
    "クローバー"
};

int drawCard(int h);

int main(){
    (1時限目のプログラムと同様なので略)
}

int drawCard (int h) {
    int draw_mark, draw_digit; // 引いたカードのマークと数
    int r; // 引いたカードの点数

    draw_mark = rand() % 4 + 1; // マークを決定する
    draw_digit = rand() % 13 + 1; // 数字を決定する

    if(h) {
        printf("%sの%s¥n", mark[draw_mark-1], digit[draw_digit-1]);
    }

    r = total[draw_digit-1];
    return r;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「7-2.c」という名前*1で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、7-2.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 7-2 7-2.c
```

4 プログラムを実行する。引いたカードのマークと数字が表示されれば成功！

【プレイヤーが引いた1枚目と2枚目の合計が21を超えた場合】

```
C:¥source>7-2.exe
【ブラックジャック】
プレイヤー一枚目：ハートのK
プレイヤー二枚目：ハートのA

ディーラー：22点　プレイヤー：21点
```

【プレイヤーが引いた1枚目と2枚目の合計が21を超えない場合】

```
【ブラックジャック】
プレイヤー一枚目：ハートの10
プレイヤー二枚目：クローバーの2
もう1枚引きますか?(y/n) > y
スペードの5
もう1枚引きますか?(y/n) > n

ディーラー：22点　プレイヤー：17点
```

解説

1 関数宣言を書く場所

今まで、関数宣言はmain関数の前、インクルード文のうしろに書いてきました。定義する自作関数が2つになった場合は、2つ続けて定義します。

【複数の自作関数を定義する】

```
#include <stdio.h>

// 自作関数 func1 の定義
void func1(void) {
    :
}
// 自作関数 func2 の定義
void func2(void) {
    :
}

//main 関数の定義
int main(){
    :
}
```

プログラムはmain関数から実行されます。よってコードを見るときは、まずはmain関数を探します。ですが自作関数がいくつもあると、下まで見ないとmain関数を見つけ出すことができません。

main関数と自作関数の定義位置を変更して、プログラムの最初にmain関数を書き、そのあとに自作関数を書くと見やすくなります。

```
#include <stdio.h>

int main(){
    int d;
    d = func1(1);
    printf("%d¥n", d);
    return 0;
}

int func1(int d) {
    return d;
}
```

このプログラムを-Wallオプションをつけてコンパイルしてみると、警告が出ます[*2]。コンパイラがこのプログラムを上から順に見ていくと、main関数の中でfunc1という名前の関数を発見しますが、そんな関数は今のところありません。よって、func1最初に出てきた時点では、「func1関数が存在しない」とコンパイラに思われてしまうのです。

ヒント

*2：C言語で用意されている関数を使うためのインクルード文がないと、同じエラーが出ます。インクルード文があると、最初に関数定義を読み込んでいるので、エラーになりません。

●コンパイラは上から順番にコードを読む

```
#include <stdio.h>

int main() {
    int d;
    d = func1(1);
    printf("%d¥n", d);
    return 0;
}

int func1(int d) {
    return d;
}
```

func1 なんて関数は知らないよ！ → 警告

＊下で定義されているが、この時点では見つかっていない

自作関数 func1 の定義

今までのように、先に自作関数func1を書いていれば、コンパイラはプログラムを上から順に見て先に自作関数func1の存在を認識しているので、そのあとにmain関数の中で自作関数func1を呼び出していても、すぐにわかるのです。

では、「main関数を最後に書くしかないのか!?」と思うかもしれませんが、コレを解決するのがプロトタイプ宣言です。

定義する関数の型と関数名、引数を書いたものをプロトタイプ宣言といいます。つまり、自作関数func1のプロトタイプ宣言は、

【プロトタイプ宣言】

```
int func1 (int d);
```

int：型　func1：関数名　(int d)：引数

となります。最後にセミコロンを忘れないでください。関数定義の1行目の最後の「{」のかわりに、セミコロンを書いたものです。

このプロトタイプ宣言をmain関数の前に書いておきます。関数が複数あるときは、それぞれのプロトタイプ宣言を書きます。

```
#include <stdio.h>

int func1(int d);
int func2(int d);

int main() {
    (略)
}

int func1(int d) {
    (略)
```

```
}
int func2(int d) {
      (略)
}
```

main関数の前にそれぞれの関数のプロトタイプ宣言をつけておくと、コンパイラは「こういう関数名を使うのね」と、先に認識してくれます。よって、main関数の中でその関数名が出てきても理解できるのです。関数の実体はmain関数のあとに書いてあるので、実行時はちゃんとその関数を実行します。

では、先ほど警告の出たプログラムにプロトタイプ宣言をつけてみます。次のようにすると、-Wallオプションをつけてコンパイルしても警告メッセージが出ることなく、問題なくコンパイルできるはずです。

```
#include <stdio.h>

int func1(int d);    // プロトタイプ宣言を追加

int main() {
(略)
```

●プロトタイプ宣言と関数

```
#include <stdio.h>

int func1(int d);

int main() {
    int d;
    d = func1(1);
    printf("%d¥n", d);
    return 0;
}

int func1(int d) {
    return d;
}
```


1時限目に作った7-1.cも、次のようにプロトタイプ宣言を使えば、

```
int drawCard(int h);
```

自作関数drawCardをmain関数のうしろに移動することができます*3。

ヒント

*3：以降、本書で作成するプログラムは、プロトタイプ宣言を使用して自作関数をmain関数のうしろに書くようにします。

2 カードのマークと数値をきめる

数値の1～4を、トランプの4種類のマークと対応させます。

(1) マークをきめる

1時限目のプログラムでは、カードの数字をrand関数を使って出しました。マークも同じ方法で出します。

```
int draw_digit; //カードの数値
int draw_mark; //カードのマーク

draw_digit = rand() % 13 + 1;
draw_mark = rand() % 4 + 1;
```

トランプのマークは4種類あるので、rand関数の値を4で割ります。

その結果が、例えば、draw_digit=8、draw_mark=1であれば、「ハートの8」のカードを引いたことになります。

draw_markは、1がハート、2がダイヤ、3がスペード、4がクローバーとします。

(2) 引いたカードのマークと数値を表示する

draw_mark=1は、コンピュータ上では数値の1としか認識されていません。これを対応するマーク名で表示しましょう。数値も同様です。例えば1のときは「A」として表示します。どちらもポインタ配列[*4]を使います。数値とマーク、それぞれの値を定義します。

```
char *digit[13] = { "A", "2", "3", "4", "5", "6", "7",
                          "8", "9", "10", "J", "Q", "K" };
char *mark[4] = {
    "ハート",
    "ダイヤ",
    "スペード",
    "クローバー"
};
```

それぞれグローバル変数として定義しています。使い方は次のとおりです。

例えばdraw_digit=12、draw_mark=1のときは、

```
printf("%sの%s", mark[draw_mark-1], digit[draw_digit-1]);
```
[*5]

で、「ハートのQ」と出力されます。これで引いたカードのマークと数値が表示できました。

ヒント

*4：引いた数は、全種類が1文字ならば普通の配列として、char digit[13]と表すことができますが、数値の10は「10」と、2文字になってしまいます。このため、ポインタ配列を使います。

ヒント

*5：配列は添え字0からはじまるので、ポインタ配列から対応する文字列を取得するには、それぞれ[引いた数-1]を添え字に指定します。

3 アドレスを使った引数の渡し方

本日のプログラムでは使用しませんが、関数の学習の仕上げとして、引数にアドレスを使う場合を紹介しておきます。

引数値を2乗する関数のプログラム*6を考えてみましょう。関数を呼び出すときの引数値を変数にします。

ヒント
*6：小さいプログラムなので、プロトタイプ宣言は使用しません。

【7-2_sample1.c】

```
#include <stdio.h>

void func1(int d1) {
    int d2 = d1 * d1;
    printf("%d¥n", d2);
}

int main(){
    int d = 3;
    func1(d);
    return 0;
}
```

結果は、「3*3」の解である「9」が出力されます。main関数の中で渡した引数の値は、自作関数func1の中では変数d1として利用できます。

今度は、自作関数func1の中で受け取った引数の値に1をプラスし、その結果を表示するプログラムを作ります。

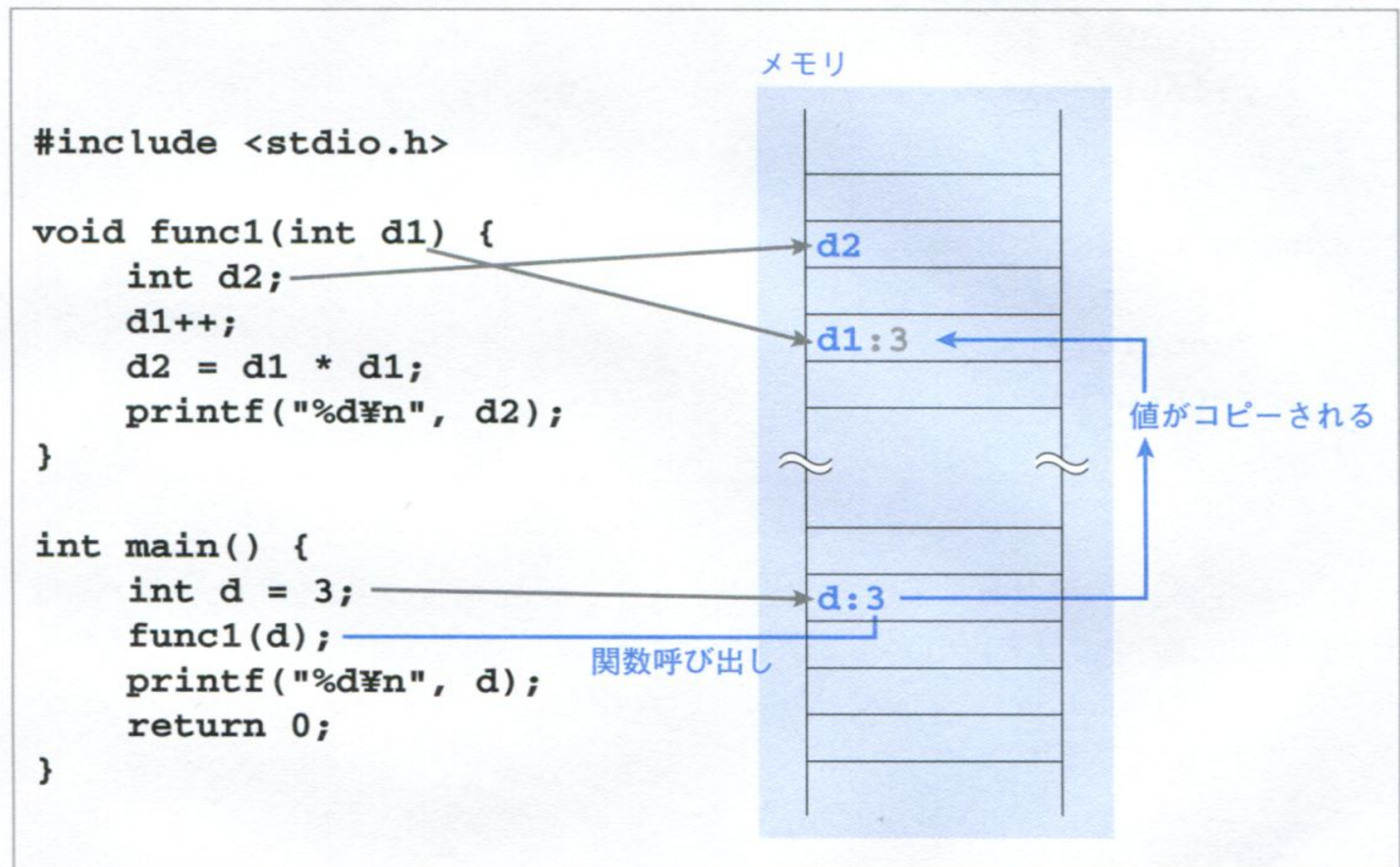
【7-2_sample2.c】

```
#include <stdio.h>

void func1(int d1) {
    int d2;
    d1++;
    d2 = d1 * d1;
    printf("%d¥n", d2);
}

int main(){
    int d = 3;
    func1(d);
    printf("%d¥n", d);
    return 0;
}
```

```
16   ←── 自作関数 func1 での出力
3    ←── main 関数での出力
```

main関数の中で自作関数func1を呼び出したときに、main関数の変数dの値が自作関数func1の変数d1にコピーされます。よって、自作関数func1の中で引数として受け取った値の変数d1の値を変更しても、main関数の変数dの値は変更されません。

●引数をコピーして渡す方法

```
#include <stdio.h>

void func1(int d1) {
    int d2;
    d1++;
    d2 = d1 * d1;
    printf("%d¥n", d2);
}

int main() {
    int d = 3;
    func1(d);
    printf("%d¥n", d);
    return 0;
}
```


値だけがコピーされるので、関数を呼び出したときの引数に指定した変数名と、関数側で受け取った変数名は同じでかまいません。別の変数として扱われます。

```
void func1(int d) {  ←── main 関数の変数 d とは別の変数
    (略)
}

int main(){
    int d = 3;
    func1(d);
    (略)
}
```

main関数の変数dと自作関数func1の変数dは、それぞれの関数のローカル変数なので、自作関数func1の変数dの値を変えても、main関数の変数dには影響がありません。しかし、

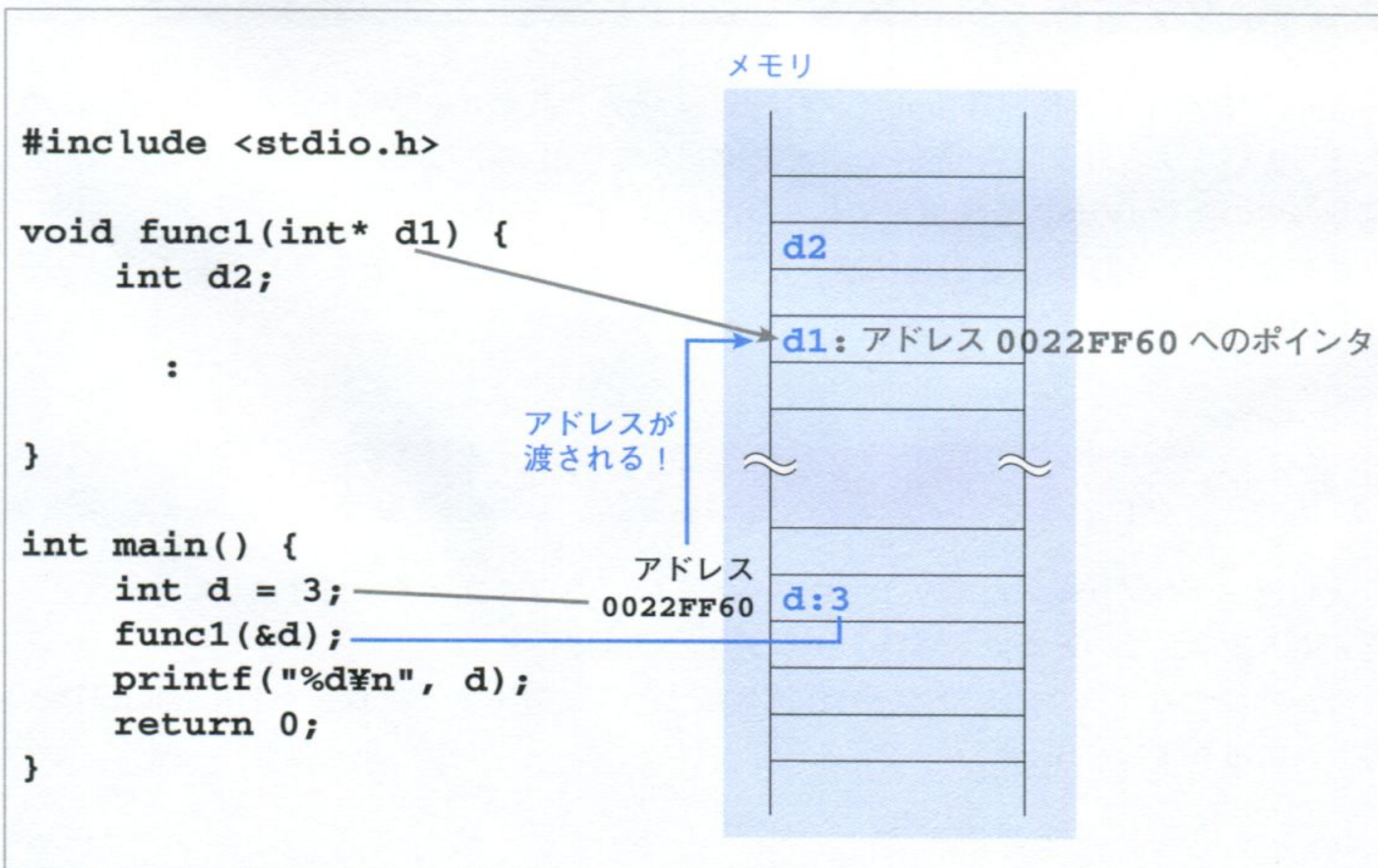
これとは逆に、引数で指定した変数の中身を変更してしまう方法があります。

今度は関数を呼び出すとき、渡す引数にアドレスを指定してみます。引数を受け取る側の関数定義では、渡ってくる引数がアドレスである場合、引数名に「*」をつけてポインタ変数として定義します。第5日に学習したポインタを思い出してください。

```
void func1(int* d1) {
      (略)
}

int main() {
      int d = 3;
      func1(&d);
      (略)
}
```

自作関数func1の変数d1は、main関数の変数dのアドレスを指しているポインタです。自作関数func1の中で引数として渡された値を使いたいときは、「*d1」と指定します。ポインタ変数d1が指しているアドレスの値を変更すると、main関数の変数dの中身が変更されます。

●引数をアドレスで渡す方法

```
#include <stdio.h>

void func1(int* d1) {
    int d2;

        :

}

int main() {
    int d = 3;
    func1(&d);
    printf("%d¥n", d);
    return 0;
}
```


引数にポインタを使って、先ほどのサンプルを作り直します。

【7-2_sample3.c】

```
#include <stdio.h>

void func1(int* d1) {
    int d2;
    (*d1)++; //d1が指すアドレスの値（main関数の変数dの値）を変更する
    d2 = *d1 * *d1;
    printf("%d¥n", d2);
}

int main() {
    int d = 3;
    func1(&d);
    printf("%d¥n", d);
    return 0;
}
```

```
16    ←――自作関数func1での出力
4     ←――main関数での出力
```

main関数の中で自作関数func1を呼び出したあとに変数dを参照すると、値が変わってしまっています。自作関数func1の中でポインタ変数d1が指しているアドレスの値（main関数の中の変数dの値）を変更してしまったからです。

このように、アドレスを引数に渡す方法を「アドレス渡し」、値をコピーして渡す方法を「値渡し」と呼びます。

まとめ

大規模なプログラムになると、プログラムをいくつかのファイルに分割して作成することがあります。このときにもプロトタイプ宣言が活躍します。

今後作成するプログラムでは、自作関数はすべてプロトタイプ宣言をしてmain関数のうしろに書きましょう。

C O L U M N

標準関数

C言語であらかじめ用意されている関数を標準関数といいます。本書で紹介した関数はそのごく一部です。その他の標準関数についてはネットや他の書籍を参考にしてください。
ここでは、その他の標準関数をいくつか紹介しておきます。

●計算を行う関数

引数や戻り値の詳細は、C:¥mingw-jp¥include¥math.hに定義されているプロトタイプ宣言を参照してください。

関数名	引数	戻り値
ceil	double x	xより小さくない最小の整数
floor	double x	xより大きくない最大の整数
fabs	double x	xの絶対値
exp	double x	eのx乗

●文字をチェックする関数

引数や戻り値の詳細は、C:¥mingw-jp¥include¥ctype.hに定義されているプロトタイプ宣言を参照してください。

関数名	戻り値
isalnum	アルファベットか数値
isalpha	アルファベット
islower	アルファベットの小文字
isupper	アルファベットの大文字
tolower	小文字を返す
toupper	大文字を返す

●計算を行う関数

引数や戻り値の詳細は、C:¥mingw-jp¥include¥stdlib.hに定義されているプロトタイプ宣言を参照してください。

関数名	引数	戻り値
abs	int n	n の絶対値(int)
atoi	const char *s	sをint型に変換
atol	const char *s	sをlong型に変換
atof	const char *s	sをdouble型に変換

2時限目で作ったプログラムに、一度引いたカードは再度引かないようにする機能をつけます。

今回作成する例題

```
C:¥source>7-3.exe
 【ブラックジャック】
プレイヤー一枚目：ハートの6
プレイヤー二枚目：スペードの7
もう1枚引きますか?(y/n) > y
クローバーの9

ディーラー：25点　プレイヤー：22点

C:¥source>7-3.exe
 【ブラックジャック】
プレイヤー一枚目：ハートの8
プレイヤー二枚目：ハートのJ
もう1枚引きますか?(y/n) > n

ディーラー：19点　プレイヤー：18点

C:¥source>
```

同じマークと数字の組みあわせを引くのは一度きり

サンプルファイルはこちら　10days_c ▶ day07-03 ▶ 7-3.c

●このレッスンのねらい

トランプのカードは4種類のマークに1～13の数字のカードが1枚ずつ、ジョーカーを除くと52枚のカードが存在します。
2時限目のプログラムではトランプのカード「○○（マーク）の△（数字）」を引くところまで作りました。実際のトランプゲームでは、同じマーク・同じ数字のカードは1枚しか存在しません。ですが、ここまでのプログラムではマークも数値も別々にランダムに出しているので、まったく同じマーク・数字の組みあわせのカードを2回以上引いてしまう可能性があります。
これではゲームとしては少々頼りないので、一度引いたカードは二度と引かないようにするしくみを作りましょう。
そのために利用するのが「2次元配列」です。配列のちょっとした応用版です。データを保存するのに便利な機能なので、この時限でぜひ理解してください。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int card[4][13]; //カードを一度引いたかどうか記録する
char total[13] = { 11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 };
char *digit[13] = { "A", "2", "3", "4", "5", "6", "7",
                        "8", "9", "10", "J", "Q", "K" };
char *mark[4] = {
    "ハート",
    "ダイヤ",
    "スペード",
    "クローバー"
};

int drawCard(int h);

int main() {
    (2時限目のプログラムと同様なので略)
}

int drawCard(int h) {
    int draw_mark, draw_digit; //引いたカードのマークと数
    int r; //引いたカードの点数

    do {
        draw_mark = rand() % 4 + 1;
        draw_digit = rand() % 13 + 1;
    } while (card[draw_mark-1][draw_digit-1]);
    card[draw_mark-1][draw_digit-1] = 1;

    if(h) {
        printf("%sの%s¥n", mark[draw_mark-1], digit[draw_digit-1]);
    }

    r = total[draw_digit-1];
    return r;
}
```

ヒント
*1：拡張子に注意して保存しましょう。

2 入力できたら、「7-3.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、7-3.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 7-3 7-3.c
```

4 プログラムを実行する。実行結果は2時限目と同じだが、一度引いたカードを引くことはなくなった

```
C:¥source>7-3.exe
```

▼

```
【ブラックジャック】
プレイヤー一枚目：ハートのK
プレイヤー二枚目：ハートのA
ディーラー：22点　プレイヤー：21点
```

```
【ブラックジャック】
プレイヤー一枚目：ハートの10
プレイヤー二枚目：クローバーの2
もう1枚引きますか?(y/n) > y
スペードの5
もう1枚引きますか?(y/n) > n
ディーラー：22点　プレイヤー：17点
```

解説

1 2次元配列

2時限目で作成したプログラムは、カードを引き続けているうちに、同じカードを再度引いてしまうことがありました。この点を改良していきましょう。

(1) 一度引いたカードを引かない方法

一度引いたカードを再び引かないプログラムを作るにはどうしたらよいでしょうか？一度引いたカードのマークと数字をおぼえておき、次からはその組みあわせが出ないようにすればよいはずです。

例えば、最初に「ハートの10」を引いたら、次からはマークが1で数字が10の組みあわせが出ないようにすればいいのでは……と考えるかもしれませんが、rand関数には特定の数字を除いた値を出す機能はありません。また、マークと数字はそれぞれrand関数を使って出し、お互いに連動していないので、この考えを実現しようとすると、さらに複雑になってしまいます。考えを改めましょう。

同じ組みあわせを出ないようにするのは不可能です。では、一度出た組みあわせをおぼえておき、もう一度同じ組みあわせを引いたら不可として、違う組みあわせになるまでカードを引き続ける方法にしましょう。

この方法の場合、「一度出た組みあわせをおぼえておく」ことが重要になります。

これは、乗り物の座席を取る場合と同じ考えで実現できます。こんな座席表があったとします。

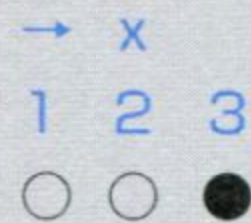
```
     → x
       1 2 3
  ↓ 1  ◎ ○ ○   ※◎：基点
  y 2  ○ ○ ○
    3  ○ ● ○
    4  ○ ○ ○
```

黒丸の所（●の席）は左上の席（◎の席）を基点として横に2、縦に3の位置として表せます。横の位置をx、縦の位置をyと呼ぶと、●は

```
x=2, y=3
```

の座席として表すことができます。

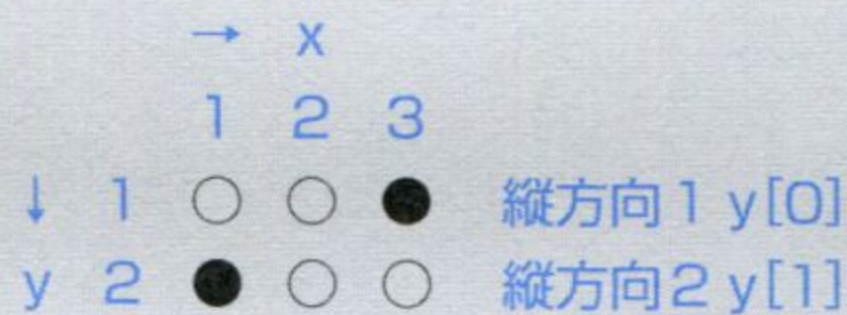
現在「x=2, y=3」の席が埋っているので、「x=2, y=3」の位置を取ろうとする他の人がいれば、そこは埋っているので他の座席を再び指定してもらいます。埋った席は●で表します。

席取りが進み、空いたところが次々と埋ってきたとします。

```
     → x
       1 2 3
  ↓ 1  ○ ○ ●
  y 2  ● ○ ●
    3  ○ ● ●
    4  ○ ○ ○
```

人がこの座席表を目で見たとき、●の部分はダメで○の部分は空いている、とひと目でわかります。プログラムでも同じようにこの座席表を表すマップを用意しておき、空いている／空いていないを記録しておく機能を作りましょう。

(2) 座席表マップを作る

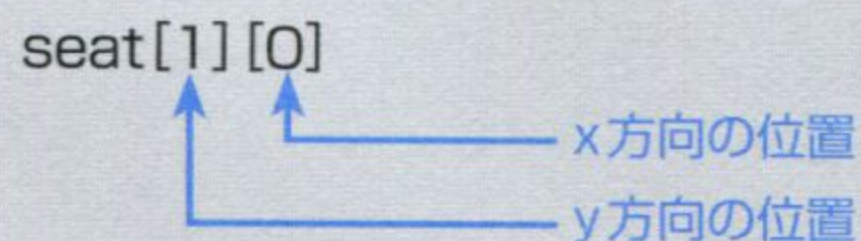
まず、座席が横に一列しかない場合を考えます。

```
→ x
1 2 3
○ ○ ●
```

座席はx方向に1、2、3だけで選ぶことができます。並んでいるので配列を使って表現できます。現在の●の部分は、x[2]*2として表せます。

ヒント
*2：添え字は0からはじまるので、左から3番目の座席の添え字は2になります。

このように1方向しかない配列を1次元配列といいます。今まで学習してきた配列は、すべて1次元配列です。

次に座席を縦に2列に増やします。

```
      → x
      1 2 3
↓ 1  ○ ○ ●   縦方向1 y[0]
y 2  ● ○ ○   縦方向2 y[1]
```

これは、先ほどのx方向のみの1次元配列が、縦に2つ用意された状態になります。y方向も1、2と並んでいるので、こちらも配列で表せます。縦の配列をy[]とします。「x=1, y=2」の位置は、

```
x[0], y[1]
```

と2つの配列を使って指定します。これをまとめて表現したものが、2次元配列です。縦方向と横方向を一度で指定できる方法です。

ここでは、座席を表す2次元配列名をseatとします。「x[0], y[1]」の位置は、

```
seat[1][0]
     ↑  ↑
     |  └──── x方向の位置
     └─────── y方向の位置
```

と、配列の添え字が2つに増えた状態で表示できます。y方向の位置を先に書くことに注意してください。

●2次元配列seatのイメージ

	1	2	3	4
1	seat[0][0]	seat[0][1]	seat[0][2]	
2	seat[1][0]	seat[1][1]	seat[1][2]	
3	seat[2][0]	seat[2][1]	seat[2][2]	
4	seat[3][0]	seat[3][1]	seat[3][2]	
5				

（→X方向、↓Y方向）

COLUMN

多次元配列

実は、もうひとつ配列の添え字が増えて、z方向も考えた3次元配列も存在します。考え方は2次元配列の応用なのでむずかしくありません。3次元配列では、

```
data[z方向の位置][y方向の位置][x方向の位置]
```

と指定します。
また、2次元配列や3次元配列を多次元配列と呼びます。

② 2次元配列の使い方

2次元配列の宣言と使用方法は、1次元配列と同様です。ただ、場所の指定はx方向とy方向の2つをセットで行います。配列のそれぞれの要素には、宣言した配列の型のデータを保存します。例えば、int型のデータを扱う横方向に3、縦方向に4の2次元配列の宣言は、次のようになります。

```
int seat[4][3];
```

初期値の設定は、y方向の配列を{ }で括ってセットにし、それをカンマ「,」でつなげます。

```
int seat[4][3] = {  { 1, 2, 3 },
                    { 4, 5, 6 },
                    { 7, 8, 9 },
                    { 10, 11, 12 } };
```

また、例えばseat[0][2]に数値1を代入するときは、次のように書きます。

```
seat[0][2] = 1;
```

3 トランプのカードを2次元配列にする

2次元配列を使ってトランプのカードを表現してみましょう。

→x	カードの数値
↓ ハートの	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
y ダイヤの	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
スペードの	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
クローバーの	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

これがすべてのカードです。つまり、y方向をカードのマーク、x方向をカードの数値とする2次元配列ができあがります。配列にはデータを格納するので、「このカードは既に引いている」場合は1、「まだ引いていない」場合は0を保存します。つまり、この2次元配列は「カードを一度引いたか否か」という情報を保存するint型の配列になります。

保存するデータは整数値なのでint型で宣言します*3。

ヒント

*3：2次元配列はint型だけでなく他のデータ型でも作成できます。

```
int card[4][13];
```

これで、トランプの2次元配列ができあがりました。これもグローバル変数として定義しましょう。

この配列を使い、カードを引く自作関数drawCardを改良してみます。今までのプログラムでは、自作関数drawCardの中でマークと数をそれぞれ一度だけ、rand関数を使って決定していました。今度は、出たカードのマークと数がすでに引かれているかどうかを確認し、もし同じ組みあわせをすでに引いていたら（そのカードはもう引けないので）、カードを繰り返し引き続けます。

「このカードはすでに引いている」場合、

```
card[draw_mark-1] [draw_digit-1]  == 1
```

となるので、これを繰り返し条件とします。

まだ一度も引いていない組みあわせが出てきたら、そのカードを引いたカードとして決定し、すでに引いたという記録を2次元配列cardに記録します。

```
card[draw_mark-1][draw_digit-1] = 1;
```

●同じ組みあわせは引かない

A 2 3 4 5 6 7 8 9 10 J Q K

①ハートの６を引く
②クローバーの３を引く
③ハートのＪを引く
④ハートの６を引く
すでに引いているので
次のカードを引く

これを、繰り返し構文のdo～while文を使って書きます。

```
do {
    draw_mark = rand() % 4 + 1;
    draw_digit = rand() % 13 + 1;
} while (card[draw_mark-1][draw_digit-1]);*4
card[draw_mark-1][draw_digit-1] = 1;
```

ヒント

*4：条件は
while (card[draw_mark-1][draw_digit-1] == 1);
と書いても同じです。2次元配列cardはグローバル変数として定義したので、初期値はすべて0で、すでに1回引いたカードの場所だけ、1になります。

これで、一度引いたカードを再度引くことはありません。
プログラムを何度か実行して、同じカードが出現しないことを確認しましょう。

まとめ

2次元配列も今後のゲームプログラムの作成で頻繁に出てきます。y方向とx方向を間違えないように使いましょう。

【ブラックジャックゲームを作ろう④】ブラックジャックゲームを完成させよう

ブラックジャックゲームのプログラムを完成させましょう。

今回作成する例題

```
C:¥source>blackjack.exe
【ブラックジャック】
ディーラー一枚目：クローバーのK
他は伏せる

プレイヤー一枚目：クローバーの4
プレイヤー二枚目：ハートのA
11として計算しますか？(y/n) > y
もう1枚引きますか?(y/n) > y
スペードのK

ディーラー：17点　プレイヤー：25点
ディーラーの勝ち！

C:¥source>
```

ブラックジャックゲームを実行

サンプルファイルはこちら　10days_c ▶ day07-04 ▶ blackjack.c

●このレッスンのねらい

3時限目までのプログラムで、ブラックジャックゲームの基本はほぼ完成しました。あとはプレイヤーが「A」のカードを引いたときに、1と11、どちらで計算するかの判定、ディーラーの1枚目を引く処理、最後に両者のカードを比べて勝敗を判定する機能をつけて完成です。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int card[4][13];
char total[13] = { 11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 };
char *digit[13] = { "A", "2", "3", "4", "5", "6",
                    "7", "8", "9", "10", "J", "Q", "K" };
char *mark[4] = {
    "ハート",
    "ダイヤ",
    "スペード",
    "クローバー"
};

int drawCard(int h);

int main() {
    int dealer; //ディーラーの引いたカードの合計
    int player; //プレイヤーの引いたカードの合計
    int draw_digit; //引いたカードの数
    int draw_mark; //引いたカードのマーク
    char y_n; //カードを引くか引かないかの答え

    srand(time(NULL));
    printf("【ブラックジャック】¥n");

    //ディーラーが引く
    draw_mark = rand() % 4 + 1;
    draw_digit = rand() % 13 + 1;
    card[draw_mark-1][draw_digit-1] = 1;
    dealer = total[draw_digit-1];
    printf("ディーラー一枚目：%sの%s¥n",
            mark[draw_mark-1], digit[draw_digit-1]);
    printf("他は伏せる¥n¥n");

    //プレイヤーが引く
    printf("プレイヤー一枚目：");
```

```
    player = drawCard(1);
    printf("プレイヤー二枚目：");
    player += drawCard(1);

    //ディーラー２枚目以降
    do{
        dealer += drawCard(0);
    } while(dealer <= 16);

    //プレイヤー３枚目以降
    while(player < 21) {
        printf("もう１枚引きますか？(y/n) > ");
        scanf("%c" , &y_n);
        while (getchar() != '¥n') {  }
        if(y_n == 'y') {
            player += drawCard(1);
        } else if (y_n == 'n') { break; }
    }

    printf("¥nディーラー：%d点 プレイヤー：%d点¥n", dealer, player);

    //勝敗の判定
    if ((dealer <= 21 && player > 21)
                || (dealer <= 21 && dealer > player)) {
        printf("ディーラーの勝ち！¥n");
    } else if ((player <= 21 && dealer > 21)
                || (player <= 21 && player > dealer)) {
        printf("プレイヤーの勝ち！¥n");
    } else {
        printf("引き分け¥n");
    }
    return 0;
}

int drawCard(int h) {
    int draw_mark, draw_digit; //引いたカードのマークと数
    int r; //引いたカードの点数
    char y_n; //カードを11として計算するかどうかの答え

    do {
        draw_mark = rand() % 4 + 1;
        draw_digit = rand() % 13 + 1;
    } while (card[draw_mark-1][draw_digit-1]);
    card[draw_mark-1][draw_digit-1] = 1;
```

```
    if(h) {
        printf("%sの%s¥n", mark[draw_mark-1], digit[draw_digit-1]);
    }

    if(h && draw_digit == 1) { //プレイヤーターンで引いた数が1のとき
        do {
            printf("11として計算しますか？ (y/n) > ");
            scanf("%c" , &y_n);
            while (getchar() != '¥n') {  }

            if(y_n == 'y') { r = total[draw_digit-1];  }
            else if (y_n == 'n') { r = 1;  }
        } while(!(y_n == 'y' || y_n == 'n'));
    } else { r = total[draw_digit-1]; }
    return r;
}
```

2 入力できたら、「blackjack.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント
*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、blackjack.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o blackjack blackjack.c
```

4 プログラムを実行する。ルールどおりにゲームが実行されれば成功！

```
C:¥source>blackjack.exe
```

▼

```
【ブラックジャック】
ディーラー一枚目：ダイヤの7
他は伏せる

プレイヤー一枚目：スペードのA
11として計算しますか？ (y/n) > y
プレイヤー二枚目：クローバーの5
```

```
もう1枚引きますか?(y/n) > n

ディーラー:18点 プレイヤー:16点
ディーラーの勝ち!
```

ディーラーの方が21に近いので、ディーラーの勝ち!

```
【ブラックジャック】
ディーラー一枚目:ダイヤのK
他は伏せる

プレイヤー一枚目:ハートの5
プレイヤー二枚目:ハートのJ
もう1枚引きますか?(y/n) > n

ディーラー:22点 プレイヤー:15点
プレイヤーの勝ち!
```

ディーラーは21より大きくなってしまったので、プレイヤーの勝ち!

解説

1 「A」のカードが出たときの判定

ブラックジャックを完成させるのに、あとひとつ、重要な機能を追加しなければなりません。これまではプレイヤーがエース「A」を引いたときは、すべて11として計算していましたが、これを1と11のどちらで計算するのかを選択する機能が必要です。

カードを引く自作関数drawCardの中を変更します。自作関数drawCardは、無条件に各カードの点数を戻り値としていました。今度はプレイヤーが「A」のカードを引いた場合と、それ以外で場合分けして戻り値を決定します。

プレイヤーが「A」のカードを引いたら、まずは「A」を11として計算するかどうか聞きます。「y」を選択した場合は11を返します。「n」を選択した場合は1を返します。答えがもし「y」でも「n」でもなかったら、もう一度プレイヤーに聞きます。つまり、プレイヤーが「y」か「n」どちらかを選択するまで繰り返し聞きます。

```
char y_n; // カードを11として計算するかどうかの答え

if(h && draw_digit == 1) { // プレイヤーターンで引いた数が1のとき
    do {
        printf("11として計算しますか? (y/n) > ");
        scanf("%c" , &y_n);
        while (getchar() != '¥n') {  }
        if(y_n == 'y') { r = total[draw_digit-1];  }
        else if (y_n == 'n') { r = 1;  }
    } while(!(y_n == 'y' || y_n == 'n'));
} else { r = total[draw_digit-1]; }
```

カードを引いたのがディーラーのときと、プレイヤーが「A」以外を引いたときは、今までと同じ処理を行います。

2 ディーラーの１枚目を引く処理

完成版のプログラムのカードの引き方は、次のようになっています。

●ブラックジャックゲームのカードの引き方

①ディーラーが１枚引く（カードを見せる）
②プレイヤーが２枚引く（カードを見せる）
③ディーラーは合計が16以下の時に引き続ける（カードを見せない）
④プレイヤーは21に近づくよう引き続ける（カードを見せる）

今までのプログラムでは、ディーラーが1枚目を引く部分は省略していました。なぜかというと、カードを引く自作関数drawCardでは、ディーラーが引いた場合の処理と、プレイヤーが引いた場合の処理は異なっています。ディーラーが引いたときはカードの種類を表示しません。ですが、ルールではディーラーの1枚目だけは表示することになっています。

これを実現しようとすると、自作関数drawCardではイレギュラーな処理が発生してしまいます。

よって、ディーラーの1枚目は自作関数drawCardを使わずに、main関数の中で書く必要があります。

ディーラーの1枚目を引く処理は、自作関数drawCardを応用します。すでに一度引いたかどうかのチェックをする必要はないので、引いたカードを表示して終了します。

```
draw_mark = rand() % 4 + 1;
draw_digit = rand() % 13 + 1;
card[draw_mark-1][draw_digit-1] = 1;
dealer = total[draw_digit-1];
printf("ディーラー一枚目:%sの%s¥n", mark[draw_mark-1], digit[draw_digit-1]);
printf("他は伏せる¥n¥n");
```

3 勝敗の判定

最後に、勝敗を判定する機能をつけましょう。ディーラーが勝利するのは、「ディーラーの点数が21以下でプレイヤーの点数が21より大きいとき」、または「ディーラーの点数が21以下でプレイヤーの点数よりも大きいとき」です。

また、プレイヤーが勝利するのは、「プレイヤーの点数が21以下でディーラーの点数が21よりも大きいとき」、または「プレイヤーの点数が21以下でディーラーの点数よりも大きいとき」です。

それ以外は引き分けとなります。これをプログラムで書いてみましょう。

```
if ((dealer <= 21 && player > 21)
            || (dealer <= 21 && dealer > player)) {
    printf("ディーラーの勝ち！¥n");
} else if ((player <= 21 && dealer > 21)
            || (player <= 21 && player > dealer)) {
    printf("プレイヤーの勝ち！¥n");
} else {
    printf("引き分け¥n");
}
```

この処理はif～else文を使って書きます。それぞれ2つの条件のどちらかを満たせば真になるので、論理演算子「||」を使います。条件文が長くなるときは、無理に1行で書かずに、適当な場所で改行するとプログラムが見やすくなります。

グローバル変数とプロトタイプ宣言は、関数の外に出して書けば完成です。

まとめ

今日のレッスンは、とくに新しい項目についての学習はありませんでした。しかし、プレイヤーの入力、コンピュータの判断、最後に勝敗決定と、ゲームプログラムの基本要素が全て入った、今までの学習のまとめのようなプログラムになったと思います。

残りはあと3日です。気を引き締めて学習を進めましょう。

練習問題

Q

blackjack.cを次のように改造しなさい。

[条件]
自作関数drawCardに渡す引数を次のとおりに変更し、ディーラーが1枚目を引く処理も自作関数drawCardを使用する。

0　ディーラー2枚目以降
1　プレイヤーターン
2　ディーラー1枚目

解答は巻末に

第 8 日

ウォーキング日記を作ろう

1時限目　構造体を理解しよう

2時限目　ウォーキング日記を書こう

3時限目　ウォーキング日記を完成させよう

本日はウォーキング日記を作ります。
1日に歩いた距離とコメントを月ごとにファイルに保存し、最新月のデータを見ることができます。ただ距離だけを測定してもつまらないので、今まで歩いた距離の合計を算出し、日本一周ウォーキングをバーチャルで行います。

今日作るプログラムについて

ウォーキング日記

ウォーキング日記は、1日の歩いた距離とその日のコメントを、月ごとのファイルに記録します。
プログラム実行時に-viewオプションをつけて今月のデータを参照することもできます。その際、今まで歩いた距離の合計を算出します。その距離からバーチャル日本一周ウォーキングを行います。
東京を出発してから今現在までどこまで歩いたかを、データから算出して表示します。

プログラムを実行するディレクトリの直下にdatというディレクトリを作成し、そこに日記のデータファイルを保存します。

●ウォーキング日記プログラムのディレクトリ構造

```
diary.exe（ウォーキング日記プログラム）
dat¥（日記のログデータ保存ディレクトリ）
  ├─ walk200901（2009年1月の記録）
  ├─ walk200902（2009年2月の記録）
  ├─ walk200903（2009年3月の記録）
  │          ：
```

データファイル名は「walk」のあとに年月を4桁2桁で表示したものとします。

ウォーキング日記プログラムの実際の動作

●日記データの追加

1 **日記プログラムを実行し、記録するデータの日付が今日の場合は「y」を入力して、[Enter] キーを押す**

```
C:¥source>diary.exe
2009/4/15 の記録をしますか？ (y/n) > y
本日の距離は？ >
```

違う日付の場合は「n」を入力し、続けて日付を、年月日のスペース区切りで入力する

```
C:¥source>diary.exe
2009/4/15 の記録をしますか？ (y/n) > n
記録する日付は？ (例：2009 4 15) > 2009 4 14
本日の距離は？ >
```

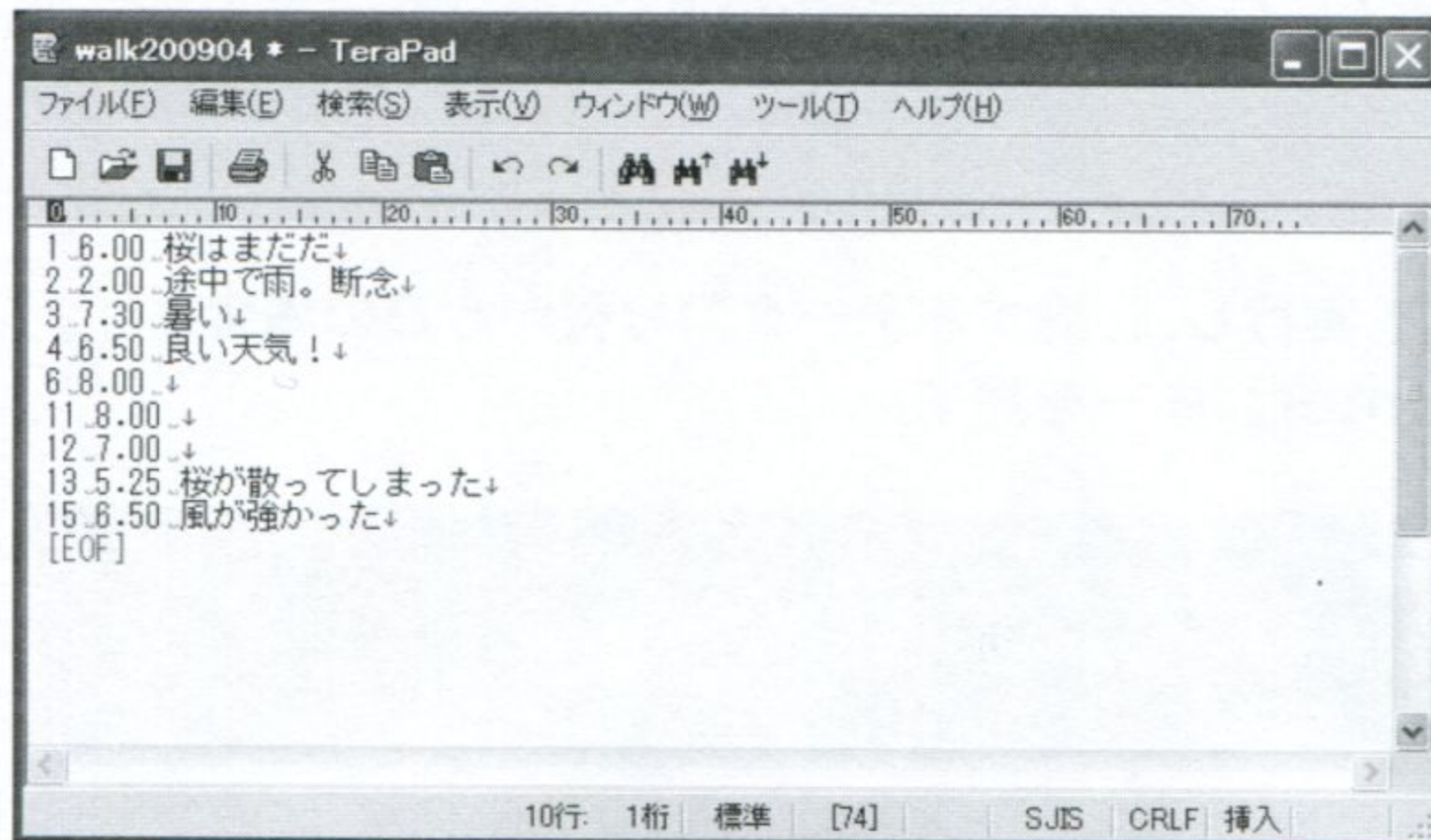
日付がおかしい場合は、そこでプログラムが終了する

```
記録する日付は？ (例：2009 4 15) > 2009 4 32
日付が正しくありません
```

2 **次に歩行距離（距離の単位はkm）とコメントを入力して、[Enter] キーを押す**

```
本日の距離は？ > 6.5
コメントは？ > 風が強かった
記録しました
```

●<参考>データの保存先のファイルをテキストエディタで開いた

```
walk200904 * - TeraPad
ファイル(F) 編集(E) 検索(S) 表示(V) ウィンドウ(W) ツール(T) ヘルプ(H)
1 6.00 桜はまだだ
2 2.00 途中で雨。断念
3 7.30 暑い
4 6.50 良い天気！
6 8.00
11 8.00
12 7.00
13 5.25 桜が散ってしまった
15 6.50 風が強かった
[EOF]
10行: 1桁 標準 [74] SJIS CRLF 挿入
```

●日記データの表示

1 日記プログラムを-viewオプションをつけて実行すると、今月のデータが表示される

```
C:¥source>diary.exe -view
```

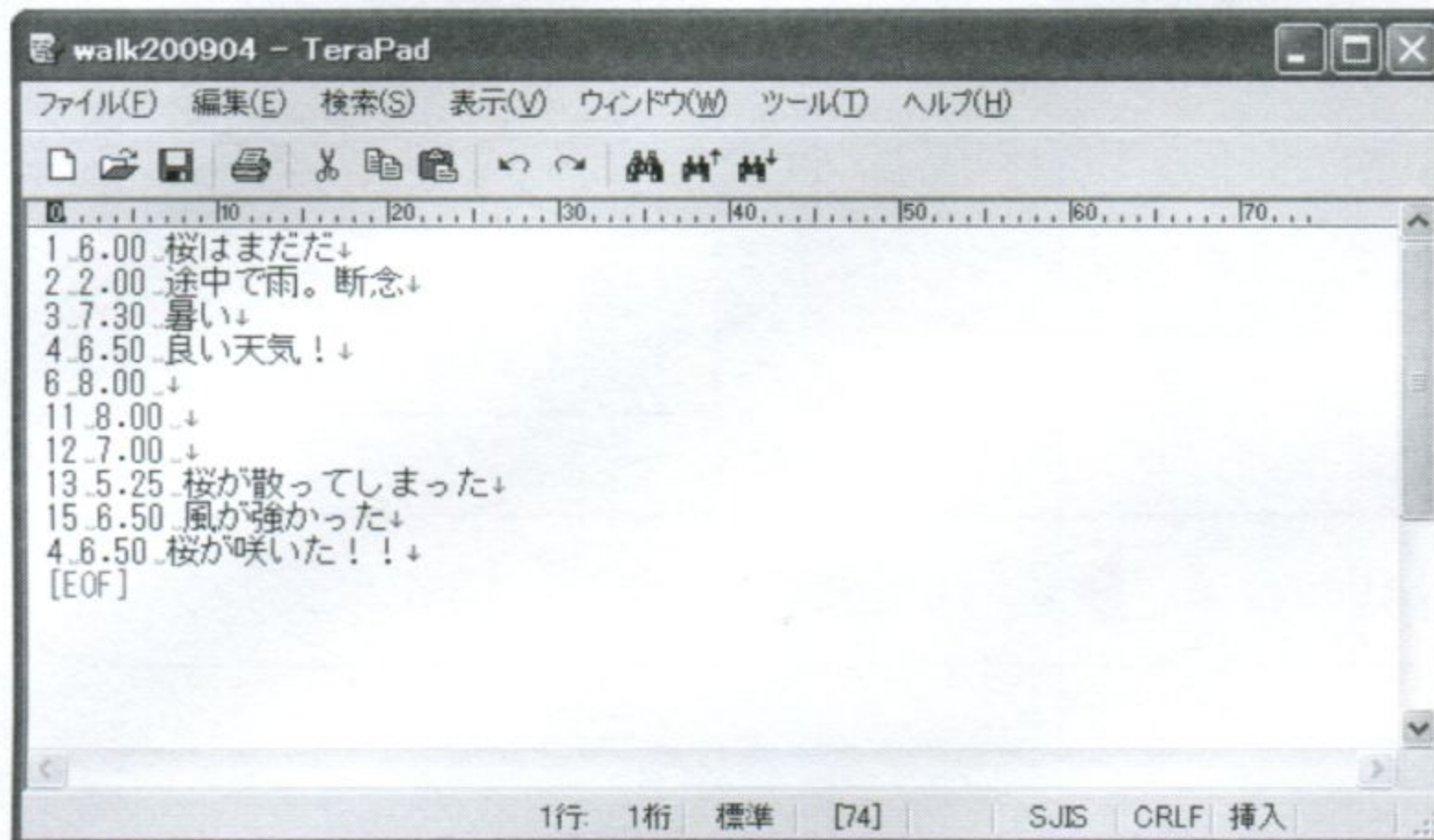
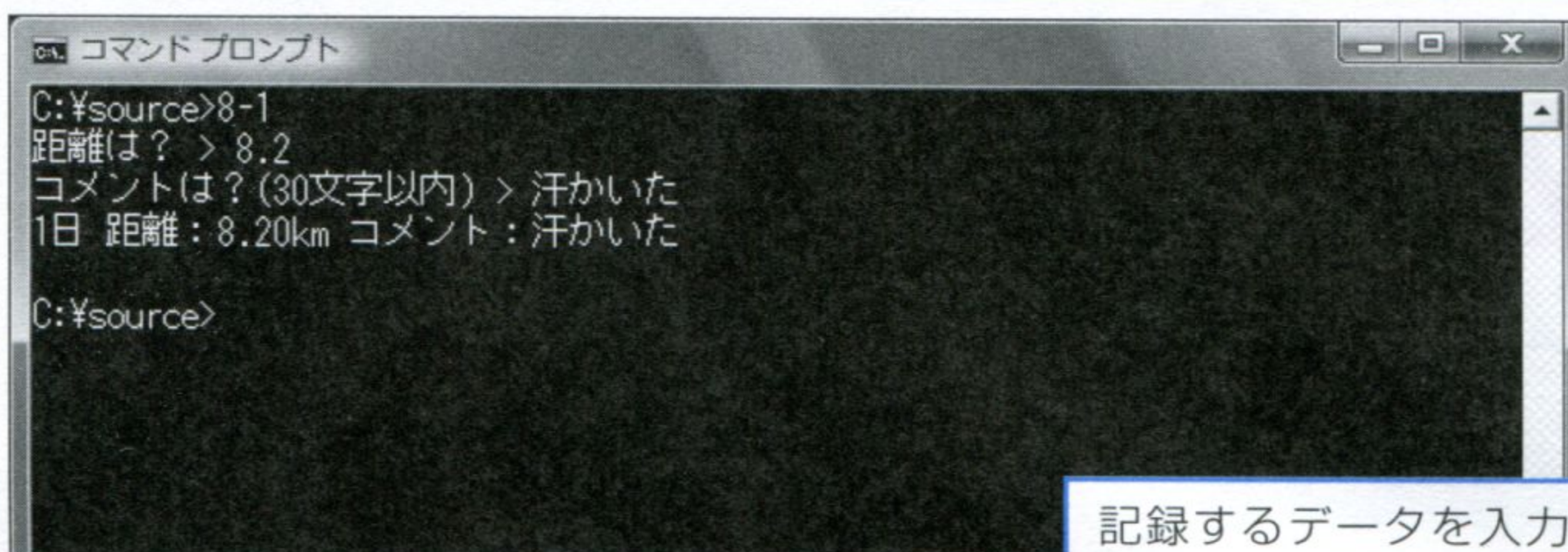
```
【2009年4月のデータ】
41.20km 千葉から1.20km地点 仙台まで588.80km
       歩距離    月累計     全累計
 1 |    6.00     6.00     47.20 桜はまだだ
 2 |    2.00     8.00     49.20 途中で雨。断念
 3 |    7.30    15.30     56.50 暑い
 4 |    6.50    21.80     63.00 良い天気！
                (略)
12 |    7.00    44.80     86.00
13 |    5.25    50.05     91.25 桜が散ってしまった
14 |
15 |    6.50    56.55     97.75 風が強かった
                (略)
30 |
97.75km 千葉から57.75km地点 仙台まで532.25km
```

・毎日の距離と月累計、記録をはじめてからの全累計とコメントが一覧で表示される
・月の頭とおわりには、そのときの累計と特定地点からの距離、次の特定地点までの距離数も表示

今月のデータがない場合は、その旨が表示される

```
【2009年6月のデータ】
今月のデータはまだありません
```

2 もし同じ日付のデータがあった場合は、あとから入力したデータが表示される

```
walk200904 - TeraPad
ファイル(F) 編集(E) 検索(S) 表示(V) ウィンドウ(W) ツール(T) ヘルプ(H)
1 6.00 桜はまだだ↓
2 2.00 途中で雨。断念↓
3 7.30 暑い↓
4 6.50 良い天気！↓
6 8.00 ↓
11 8.00 ↓
12 7.00 ↓
13 5.25 桜が散ってしまった↓
15 6.50 風が強かった↓
4 6.50 桜が咲いた！！↓
[EOF]
1行 1桁 標準 [74] SJIS CRLF 挿入
```

```
【2009年4月のデータ】
41.20km 千葉から1.20km地点 仙台まで588.80km
      歩距離   月累計    全累計
 1 |   6.00    6.00    47.20 桜はまだだ
              (略)
 4 |   6.50   21.80    63.00 桜が咲いた！！
              (略)
97.75km 千葉から57.75km地点 仙台まで532.25km
```

3 途中で特定地点を通過した場合は、コメントの横にその旨を表示する

```
29 | 20.00  656.55   697.75 今日も暑い！  (仙台到着 札幌まで2102.25km ！)
```

第8日

1時限目

構造体を理解しよう

【ウォーキング日記を作ろう❶】

この時間は構造体について学習し、ウォーキング日記データの基礎を作成しましょう。

今回作成する例題

```
コマンド プロンプト
C:¥source>8-1
距離は？ > 8.2
コメントは？(30文字以内) > 汗かいた
1日 距離：8.20km コメント：汗かいた

C:¥source>
```

記録するデータを入力して、それを標準出力する

サンプルファイルはこちら 10days_c ▶ day08-01 ▶ 8-1.c

●このレッスンのねらい

ウォーキング日記は月ごとにファイルに記録します。その内容は、日ごとに「日付」「1日の歩行距離」「コメント」の3つです。このデータを月ごとに読み込んだ場合、各日付、歩行距離、コメントをそれぞれ別の配列として持っておくこともできますが、まとまったデータとして保存しておくと、プログラムで利用しやすくなります。
今回使うウォーキング日記のデータは、日付は整数、歩行距離は実数、コメントは文字列と、データ型が異なるので多次元配列は使えません。
しかし、C言語では異なる型のデータをひとまとめにして扱うことのできる方式が用意されています。それを「構造体」といいます。この時限では構造体の使い方をマスターします。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>

typedef struct {
    int day;
    double dist;
    char comment[32];
} diary;

int main(int argc, char* argv[]) {
    diary today_data;

    printf("距離は？ > ");
    scanf("%lf", &today_data.dist);
    while (getchar() != '¥n') { }
    printf("コメントは？(30文字以内) > ");
    fgets(today_data.comment, 32, stdin);

    today_data.day = 1;

    printf("%d日 距離：%.2lfkm コメント：%s",
        today_data.day, today_data.dist, today_data.comment);
    return 0;
}
```

2 入力できたら、「8-1.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、8-1.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 8-1 8-1.c
```

ヒント

*1：拡張子に注意して保存しましょう。

4 プログラムを実行する

```
C:¥source>8-1.exe
```

▼

```
距離は？ > 8.2 ←距離を入力
コメントは？（30文字以内） > 今日は雨が降ってきた←コメントを入力
1日 距離：8.20km コメント：今日は雨が降ってきた←入力した距離とコメント
                                            が表示されれば成功！！
```

解説

1 構造体とは何か？

ウォーキング日記で記録するデータとその値は、次の3種類です。

日＝1（日）*2
1日の歩行距離＝8.2（km）
コメント＝「今日は雨が降ってきた」

ヒント
*2：データは年月ごとにファイルに保存します。年月はファイル名からわかるので、日データのみ記録します。

この3種類のデータを「1日のデータ」としてまとめて保持する場合に、構造体を使います。

構造体とは、異なる型を持つデータをまとめてひとつのデータ型として扱う形です。つまり、int型やchar型と同じようにひとつのデータの型を意味します。既存の型ではなく、自分で新たに作ったデータ型ということです。

ただし、int型やchar型はC言語で用意されている型なので、何もせずにそのまま使うことができますが、構造体はそのプログラムで独自に作った型なので、プログラムの最初に型の宣言を書いておく必要があります。

構造体の宣言の書き方は、次のようにします。

【構造体の宣言】

```
struct 構造体名 {
    データ型 変数名；
    データ型 変数名；
        （略）
};
```

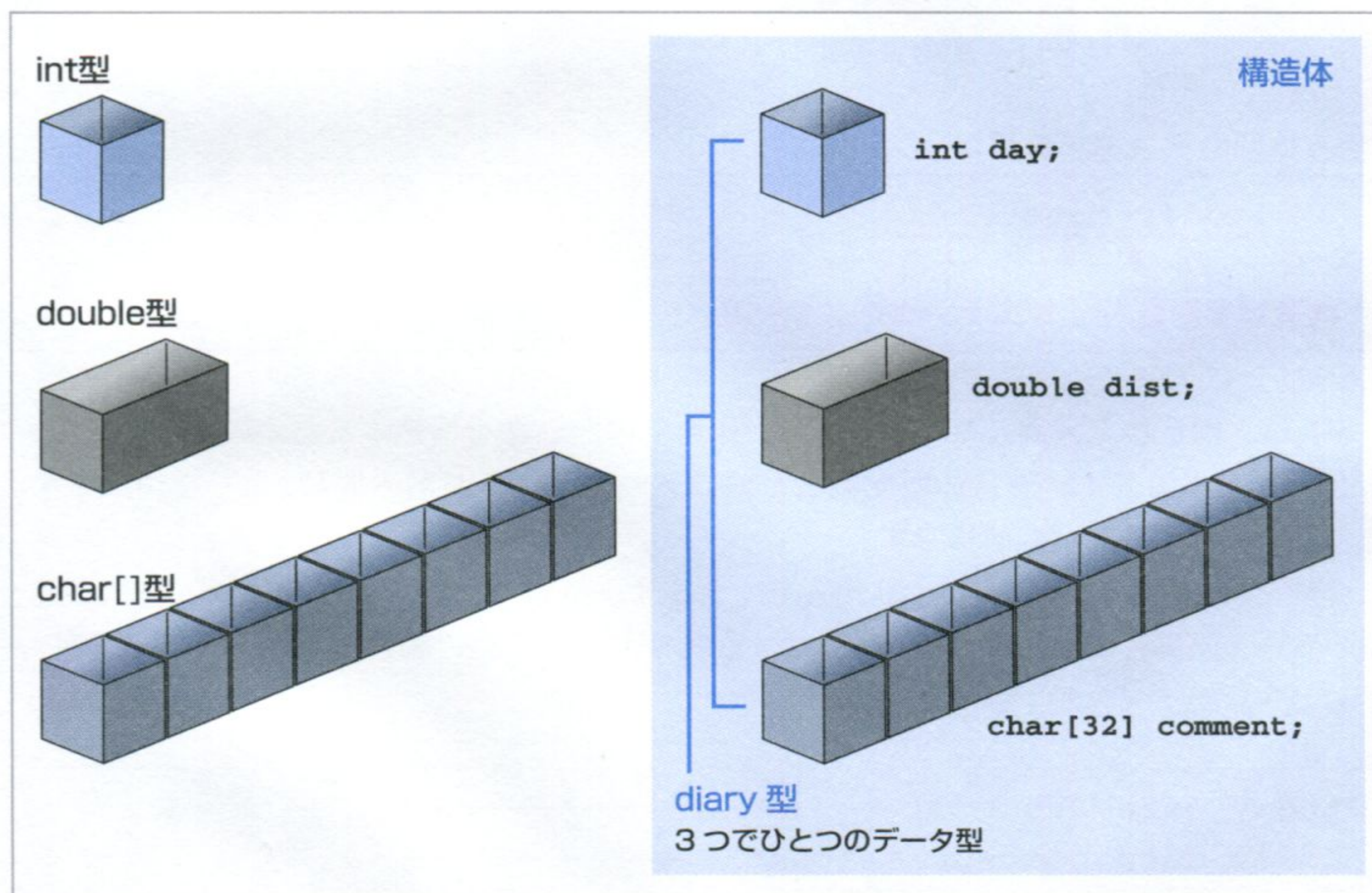
これで終了です。構造体名は、変数名などと同様に半角英数文字で自由に命名します。宣言の中には、まとめたいデータの型と変数名をすべて宣言します。これらの構造体の要素は、メンバと呼ばれます。

今回作りたいデータは、「日」「距離」「コメント」の3種類です。よって、構造体名をdiaryにした宣言は、次のとおりです。

```
struct diary{
    int day;
    double dist;
    char comment[32];
}
```

この構造体の宣言を、プログラムの最初の方に書いておきます。グローバル変数を宣言する場所と同じです。

そうすると、int型やchat型などのデータ型と同様に、diary型というデータ型がこのプログラム内で使えるようになります。

●構造体

データの型がきまれば、あとはその型を使った変数が利用できます。構造体diary型の変数を宣言しましょう。構造体もデータ型の一種なので、

【構造体の変数宣言】

```
struct diary 変数名 = { 日 , 距離 , コメント };
```

データ型（構造体名）　初期値

と宣言します。整数のデータ型は「int」で宣言しますが、構造体は「struct 構造体名」で型を宣言します。初期値データは{ }で括り、構造体diary型を構成するデータである「日」「距離」「コメント」の3つのデータを、前から順にカンマ「,」で区切って宣言します。実際にデータを入れてみましょう。

```
struct diary today_data = { 1, 8.2, "今日は雨が降ってきた"};
```

これで、構造体diary型の変数today_dataが初期化されました。変数today_dataは、データの型が異なるだけで、他の型の変数と基本的に扱い方は一緒です。

最初に宣言だけしておいて、データはあとから代入することもできるし、データを参照したり、変更したりすることもできます。

次に、構造体型のデータを参照してみます。構造体diary型の場合は3種類のデータを持つので、それぞれどのデータを対象にしたいのかを指定します。変数today_dataの「距離」を参照したい場合は、次のようにします。

【構造体型のデータを参照】

```
today_data.dist
```

構造体型変数名 . 構成データのメンバ名

distは、構造体を宣言したときに「歩行距離」のデータを格納するための変数として宣言したメンバ名です。構造体型変数名と構成データのメンバ名の間に、「.」をつけます。それぞれ、他のメンバも同様です。

データを参照するだけでなく、代入もできます。構造体の中の各変数は、int型やchar型のデータの場合はそのまま代入できます。しかし、文字列の場合は、strcpy関数を使いましょう。

```
today_data.day = 3;                          ←○
today_data.comment = "test";                 ←×
strcpy(today_data.comment, "test");          ←○
```

では、構造体diaryを宣言し、この構造体型を使う変数を作成してみます。

【8-1_sample1.c】

```
#include <stdio.h>

struct diary {
    int day;
    double dist;
    char comment[32];
};

int main() {
    struct diary today_data = { 1, 8.2, "今日は雨が降ってきた"};

    printf("%d日 距離：%.2lfkm コメント：%s",
        today_data.day, today_data.dist, today_data. comment);
    return 0;
}
```

```
1日 距離：8.20km コメント：今日は雨が降ってきた
```

構造体の、それぞれのデータの内容を出力しました。

2 構造体と構造体型変数の書き方

構造体を宣言すると同時に、その型を持つ構造体型変数も宣言できます。

【構造体と同時に構造体型変数を宣言】

```
struct 構造体名 {
    (略)
} 変数名;
```

この宣言をmain関数の外に書くと、構造体型変数はグローバル変数になります。先ほどの構造体型変数today_dataをグローバル変数にしてみます。

【8-1_sample2.c】

```
#include <stdio.h>
#include <string.h>

struct diary {
    int day;
    double dist;
```

```
    char comment[30];
} today_data;

int main() {
    today_data.day = 1;
    today_data.dist = 8.2;
    strcpy(today_data.comment, "今日は雨が降ってきた");

    printf("%d日 距離：%.2lfkm コメント：%s",
        today_data.day, today_data.dist, today_data.comment);
    return 0;
}
```

構造体の宣言と同時に構造体型変数today_dataも定義したので、main関数の中でそれぞれのデータの初期値を代入しています。

構造体やその型を持つ変数は、宣言する場所によって有効範囲が異なります。

3 typedefを使う

int型や他のデータ型の変数は「データ型 変数名;」と宣言しますが、構造体型の変数は「struct 構造体名 変数名;」で宣言され、「struct 構造体名」でひとつのデータ型のような扱いをしました。

しかし、構造体を次のようにtypedefを使って定義すると、構造体名だけでデータ型として扱うことができます。

【typedefで構造体を定義】

```
typedef struct {
    (略)
} diary;
```

diary → データ型 ＝ diary型

```
diary today_data;
```

diary → データ型　today_data → 変数名

4 構造体にデータを入力して表示してみる

この時限に作るプログラムは、構造体diary型を作成し、diary型の変数に標準入力からデータを代入します。構造体メンバのdayだけは、固定で1を設定しておきます。

構造体を使用する以外は特に新しく学習することはありませんが、コメントの入力部分だけ、注意が必要です。

fgets関数を使い、第2引数に読み込む文字数、第3引数に読み込み先を指定しています。

```
fgets(today_data.comment, 32, stdin);
```

この場合、stdin、つまり標準入力から最大「32－1＝31*3」の文字数を読み込みます。

なお、fgets関数を使う理由は、第6日2時限目にすでに説明しました。第6日2時限目のプログラムでは、文字数が指定よりも多くなった場合は内容不備としましたが、今回は同じ日付のデータを再度入力できるので、そのままとします。

ヒント

*3：31文字の入力は、半角30文字＋改行を想定しています。本日2時限目に作るプログラムでは、もし半角31文字以上を入力した場合を考えて、処理を加える必要があります。

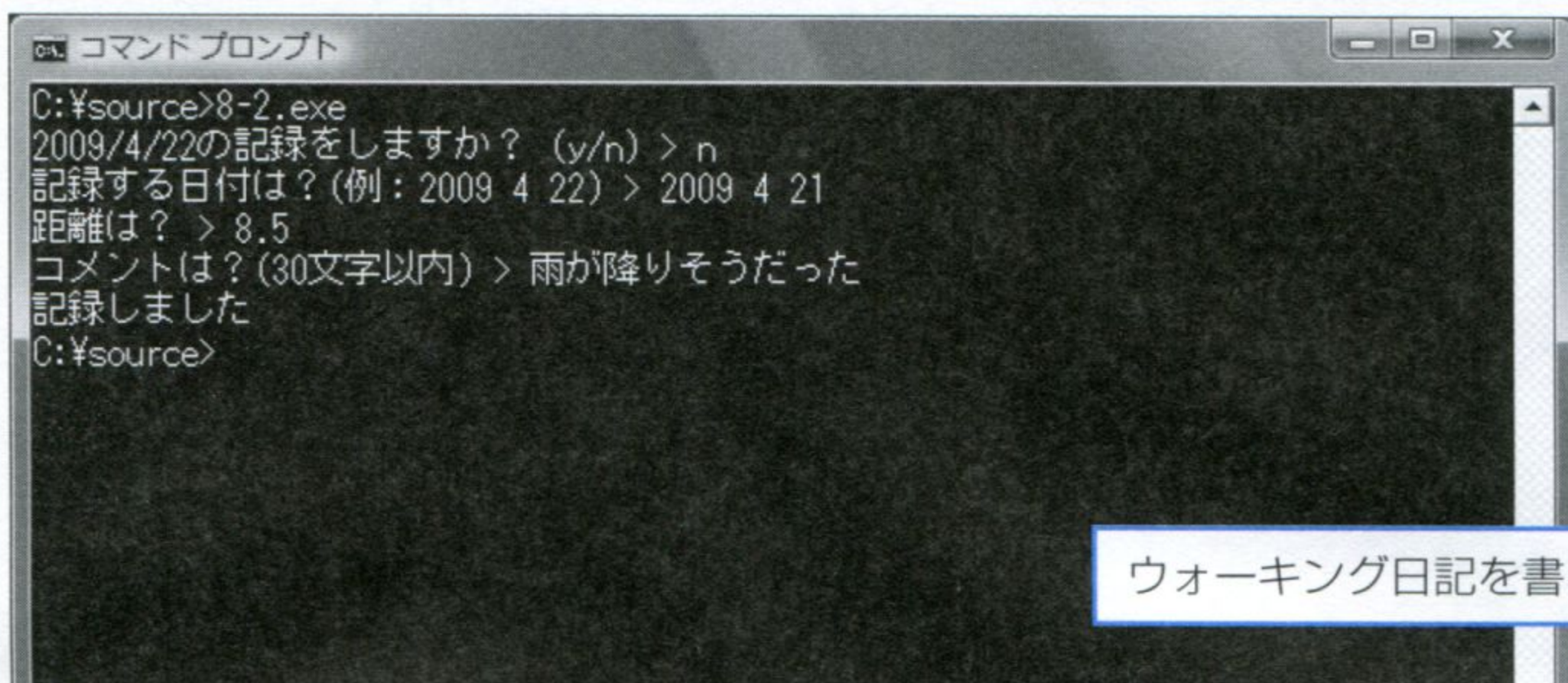
まとめ

構造体を使って新たなデータ型を作成する方法を学びました。

構造体名と変数名がごちゃごちゃにならないように、それぞれ自分でわかりやすい名前をつけましょう。

ウォーキング日記を書くプログラムを作成します。

今回作成する例題

```
C:¥source>8-2.exe
2009/4/22の記録をしますか？ (y/n) > n
記録する日付は？(例: 2009 4 22) > 2009 4 21
距離は？ > 8.5
コメントは？(30文字以内) > 雨が降りそうだった
記録しました
C:¥source>
```

ウォーキング日記を書く

サンプルファイルはこちら　10days_c ▶ day08-02 ▶ 8-2.c

●このレッスンのねらい

1時限目でウォーキング日記の構造体を作成し、日記のデータを入力するプログラムを作成しました。
この時間はそのプログラムを応用して、ウォーキング日記をデータファイルに記録するプログラムを作成します。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <string.h>
#include <time.h>

typedef struct {
    int day;
    double dist;
    char comment[32];
} diary;

void writeData(char datafile[]);
void viewData(char datafile[]);

char datadir[] = "dat/";
int year, month, day;

int main(int argc, char* argv[]) {
    char datafile[11];
    struct tm *date;
    time_t now;

    now = time(NULL);
    date = localtime(&now);
    year = date->tm_year + 1900;
    month = date->tm_mon + 1;
    day = date->tm_mday;
    sprintf(datafile, "walk%04d%02d", year, month);

    if ((argc > 1) && (strcmp(argv[1], "-view")==0)) {
        viewData(datafile);
    }
    else {
        writeData(datafile);
    }
    return 0;
}

void writeData(char datafile[]) {
```

```
diary today_data;
FILE *outfp; // 出力ファイルのファイルポインタ
char y_n = 'n';
char filename[15];
char input_str[12];

printf("%d/%d/%d の記録をしますか? (y/n) > ", year , month, day);
scanf("%c", &y_n);
while (getchar() != '¥n') { }
if(y_n != 'y') {
    printf("記録する日付は?(例:%d %d %d) > ", year, month, day);
    year = 0; month = 0; day = 0;
    gets(input_str);
    sscanf(input_str, "%d %d %d", &year, &month, &day);
    if((year > 9999) || (month > 12) || (month < 1) ||
        (day > 31) || (day < 1)) {
        printf("日付が正しくありません¥n");
        return;
    }
    sprintf(datafile, "walk%04d%02d", year, month);
}
today_data.day = day;

printf("距離は? > ");
scanf("%lf", &today_data.dist);
while (getchar() != '¥n') {  }
printf("コメントは?(30文字以内) > ");
fgets(today_data.comment, 32, stdin);
if((strlen(today_data.comment) == 31) &&
        (today_data.comment[30] != '¥n')) {
    while (getchar() != '¥n') {  }
}

sprintf(filename, "%s%s", datadir, datafile);
if((outfp = fopen(filename, "a+")) == NULL) {
    printf("ファイルオープンエラー¥n");
    return;
}
fprintf(outfp, "%d %.2lf ",
    today_data.day, today_data.dist);
if((strlen(today_data.comment) == 31) &&
        (today_data.comment[30] != '¥n')) {
    today_data.comment[30] = '¥n';
    today_data.comment[31] = '¥0';
```

```
        }
        fprintf(outfp, "%s", today_data.comment);
        fclose(outfp);
        printf("記録しました");
}

void viewData(char datafile[]){
}
```

2 **入力できたら、「8-2.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する**

ヒント
*1：拡張子に注意して保存しましょう。

3 **コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、8-2.cをコンパイルする**

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 8-2 8-2.c
```

4 **プログラムを実行するディレクトリの下に、「dat」という名前のディレクトリを作成する**

5 **プログラムを実行する。データを入力し、それがdatディレクトリにある日記データファイルに記録されていれば成功！**

```
C:¥source>8-2.exe
```

▼

```
2009/4/14の記録をしますか？ (y/n) > n
記録する日付は？(例：2009 4 14) > 2009 4 13
距離は？ > 5.25
コメントは？(30文字以内) > 桜が散ってしまった
記録しました
```

解説

1 日付を取得する

次のプログラムは、現在の日付を取得するものです。

【8-2_sample1.c】

```
#include <stdio.h>
#include <time.h>

int year, month, day;

int main() {
    struct tm *date;
    time_t now;

    now = time(NULL);
    date = localtime(&now);
    year = date->tm_year + 1900;
    month = date->tm_mon + 1;
    day = date->tm_mday;
    printf("%04d 年 %02d 月 %02d 日 ", year, month, day);
    return 0;
}
```

ここでもstruct、つまり構造体が出てきました。「struct tm」とは、C言語で定義されている暦時刻の各要素（時分秒等）を格納する構造体です。

もうひとつ新しくtime_tという型が出てきましたが、これは時刻を表す型です。

time関数[*2]で現在の時刻を取得し、それをtime_t型変数nowに格納します。さらにlocaltime関数[*2]で、引数のデータを現地時間（日本時間）に変換し、tm構造体型の変数に格納しています。

ヒント
*2：これらの関数はtime.hをインクルードして使用します。

tm構造体の構成は、time.hに定義されています。中身は次のとおりです。

```
struct tm {
    int tm_sec;      /* 秒 [0-61] 最大 2 秒までのうるう秒を考慮 */
    int tm_min;      /* 分 [0-59] */
    int tm_hour;     /* 時 [0-23] */
    int tm_mday;     /* 日 [1-31] */
    int tm_mon;      /* 月 [0-11] 0 からはじまることに注意 */
    int tm_year;     /* 年 [1900 からの経過年数 ] */
    int tm_wday;     /* 曜日 [0: 日 1: 月 ... 6: 土 ] */
    int tm_yday;     /* 年内の通し日数 [0-365] 0 からはじまることに注意 */
    int tm_isdst;    /* 夏時間が無効であれば 0 */
};
```

注意しなければならないのは、8-2_sample1.cではtm構造体の変数に「*」がついている点です。

```
struct tm *date;
```

これは、変数dateはtm構造体を指すポインタである、ということになります。

通常の構造体変数のメンバを参照するときは「.」を使いましたが、ポインタの場合は「->」を使います。次のようにすると、現在の月を取得できます。

```
date->tm_mon
```

少々面倒ですが、「現在時刻を取得するときはこのような書き方をするものだ」とおぼえてしまってください。

tm構造体に格納されたデータの年、月はそのままの数値ではないので、それぞれに加工しないと正確な数値になりません。注意してください。年は1900をプラスし、月は1をプラスする必要があります。

2 ウォーキング日記プログラムのmain関数部分を作成する

ウォーキング日記プログラムでは、-viewオプションをつけたら日記の閲覧、つけなければ日記の記録を行います

日記の閲覧と記録は、それぞれ関数化します。どちらもファイル名を引数とし、戻り値なしの関数にします。

```
void writeData(char datafile[]);      // 日記の記録関数
void viewData(char datafile[]);       // 日記の閲覧関数
```

main関数では、まず現在時刻からファイル名を作成します。

日記の閲覧の場合は、そのファイルを見るようにします。記録の場合は、そのファイルに記録します。もちろん本日以外のデータを記録することもできますが、それは自作関数writeDataの中で行います。

```
int main(int argc, char* argv[]) {
    char datafile[11];

    // 時間を取得し、ファイル名を設定する
    if ((argc > 1) && (strcmp(argv[1], "-view")==0)) {
        viewData(datafile);
    }
    else {
        writeData(datafile);
    }
    return 0;
}
```

3 データファイルに記録する自作関数writeData

この時限では、日記の記録関数を先に作成します。自作関数writeDataで行うことは、次の4点です。

・記録するデータは今日の日付で良いかどうか確認
・違う日付の場合はその日付を取得（データファイル名の設定し直し）
・記録データの入力
・データの記録

主な機能はデータの入力と記録なので、1時限目で作成した8-1.cのプログラム内容を応用して作ります。

データファイルの1行は1日ごとのデータになります。各情報は空白で区切ります。

```
日付 歩行距離 コメント
```

データファイルの実際の中身は次のようになります。

```
1 8.20 今日もよく歩いた！
2 4.50
3 4.20 少し明日痛んだので途中で断念
```

コメントはなくてもかまいません。距離の小数点以下は2桁までとします。

もしデータを間違えて訂正したい場合は、かまわずもう一度同じ日付で追加作業を行います。データファイルには同じ日付のデータが追加で書き込まれますが、読み込むときに最後のデータを優先するので、問題ありません。

データファイルは、ファイルが存在しないときに作成し、存在するときは追加書きを行う「a+」モードでオープンします。

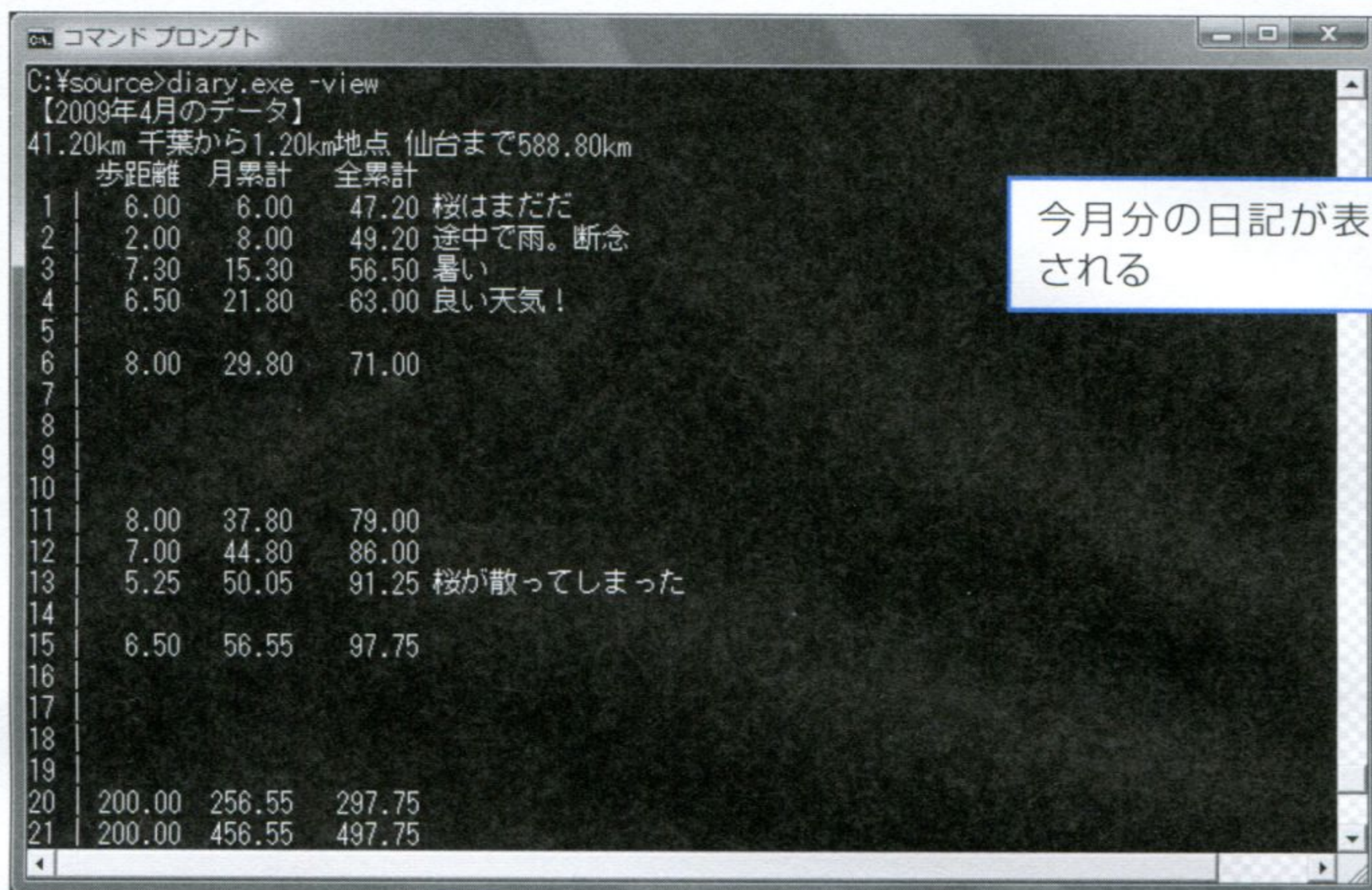
今までのプログラムと違い、データファイルは「dat」ディレクトリ*3の下に作成するので、オープンするファイルはディレクトリ名を付加して指定します。

ヒント
*3：「dat」ディレクトリはあらかじめ作成しておきます。

```
sprintf(filename, "%s%s", datadir, datafile);
if((outfp = fopen(filename, "a+")) == NULL) {
    printf("ファイルオープンエラー¥n");
    return;
}
```

ファイルへの書き出しは、もし歩行距離が数値としておかしい場合は、0.00になります。

コメントは最大30文字になるようにしたいので、31文字（30文字＋改行）を超えて入

力した場合は、最初の30文字で切るようにします。
　もし入力間違いがあった場合は、あとから同じ日付でデータを入力すれば表示時はそちらが優先されるので、詳細なチェックは省略しています。

まとめ

　日付の取得に、学習したばかりの構造体を利用しました。日付の取得方法は少々特殊ですが、よく使う技術です。ここで使い方をおぼえてしまいましょう。

第8日 3時限目

ウォーキング日記を完成させよう

【ウォーキング日記を作ろう③】

ウォーキング日記を表示する関数を作成して、ウォーキング日記プログラムを完成させましょう。

今回作成する例題

```
C:¥source>diary.exe -view
【2009年4月のデータ】
41.20km 千葉から1.20km地点 仙台まで588.80km
     歩距離  月累計   全累計
 1 |   6.00    6.00    47.20 桜はまだだ
 2 |   2.00    8.00    49.20 途中で雨。断念
 3 |   7.30   15.30    56.50 暑い
 4 |   6.50   21.80    63.00 良い天気！
 5 |
 6 |   8.00   29.80    71.00
 7 |
 8 |
 9 |
10 |
11 |   8.00   37.80    79.00
12 |   7.00   44.80    86.00
13 |   5.25   50.05    91.25 桜が散ってしまった
14 |
15 |   6.50   56.55    97.75
16 |
17 |
18 |
19 |
20 | 200.00  256.55   297.75
21 | 200.00  456.55   497.75
```

今月分の日記が表示される

サンプルファイルはこちら　10days_c ▶ day08-03 ▶ diary.c

●このレッスンのねらい

2時限目までにウォーキング日記を記録する機能を作成したので、日記を表示する機能を追加して、日記プログラムを完成させます。
また、日記のデータファイルは別のディレクトリ下にまとめて保存してあるため、ディレクトリの一覧を読み込む関数を使って、データを読み込みます。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <dirent.h>
#include <sys/types.h>
#include <stdlib.h>

typedef struct {
    (8-2.c と同じなので略)
} diary;

typedef struct {
    char name[10];
    double dist;
} dist_data;


void writeData(char datafile[]);
void viewData(char datafile[]);

char datadir[] = "dat/";
int year, month, day;

int main(int argc, char* argv[]) {
    (8-2.c と同じなので略)
}

void writeData(char datafile[]) {
    (8-2.c と同じなので略)
}

/* 日記を表示する関数 */
void viewData(char datafile[]) {
    diary walk_diary[31];
    diary tmp;
    char filename[15];
    char str[128];
    int i, j, file_c = 0;
```

```
int c = 0; //現時地点
double total = 0, m_total = 0;
dist_data map[] = {
        { "東京", 0.0 },{ "千葉", 40.0 },{ "仙台", 630.0 },
        { "札幌", 2800.0 },{ "金沢", 4100.0 },{ "博多", 5000.0 },
        { "広島", 6330.0 },{ "名古屋", 8000.0 },{ "東京", 8300.0 }};
int map_c = 8; //map最大添え字
FILE *fp; // 入力ファイルのファイルポインタ
DIR *dir;
struct dirent *dp;
char *datafiles[100];
double dist[31] = { 0.0 };    // 0で初期化する
int lastday[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

//データファイル名一覧を取得する
dir=opendir(datadir);
while((dp=readdir(dir)) != NULL) {
    if(strncmp(dp->d_name, "walk", 4) == 0) {
        datafiles[file_c] = (char*)malloc(strlen(dp->d_name) + 1);
        strcpy(datafiles[file_c++], dp->d_name);
    }
}
closedir(dir);
//歩行距離累計を算出する
for(i = 0 ; i < file_c; i++) {
    if(strcmp(datafiles[i], datafile) == 0) { continue; }
    sprintf(filename, "%s%s", datadir, datafiles[i]);
    if((fp = fopen(filename, "r")) == NULL) {
        printf("ファイルオープンエラー¥n");
        continue;
    }
    for(j = 0; j < 31; j++) { dist[j] = 0; }
    while(fgets(str, sizeof(str), fp) != NULL) {
        sscanf(str, "%d %lf %s¥n",  &tmp.day, &tmp.dist, tmp.comment);
        if(tmp.day > 0) { dist[tmp.day-1] = tmp.dist; }
    }
    fclose(fp);
    for(j = 0; j < 31; j++) { total += dist[j]; }
}

//今月のデータを取得する
sprintf(filename, "%s%s", datadir, datafile);
printf("【%d年%d月のデータ】¥n", year, month);
if((fp = fopen(filename, "r")) == NULL) {
```

```
        printf("今月のデータはまだありません¥n");
        return;
    }
    while(fgets(str, sizeof(str), fp) != NULL) {
        strcpy(tmp.comment, "");
        sscanf(str, "%d %lf %s¥n",  &tmp.day, &tmp.dist, tmp.comment);
        if(tmp.day > 0) {
            walk_diary[tmp.day-1].day = tmp.day;
            walk_diary[tmp.day-1].dist = tmp.dist;
            strcpy(walk_diary[tmp.day-1].comment, tmp.comment);
        }
    }
    fclose(fp);

    for(j = map_c; j > 0; j--) {
        if(total >= map[j].dist) { c = j; break; }
    }
    printf("%.2lfkm %sから%.2lfkm地点",
        total, map[c].name, total - map[c].dist);
    if(c < map_c) { printf(" %sまで%.2lfkm",
                map[c+1].name, map[c+1].dist - total); }
    printf("¥n       歩距離   月累計    全累計¥n");

    //今月のデータを表示する
    if((month==2) && ((year%4==0 && year%100!=0) || year%400==0)) {
        lastday[1] = 29; }
    for(i = 0; i < lastday[month-1]; i++) {
        printf("%2d |", i + 1);
        if((i + 1) == walk_diary[i].day) {
            m_total += walk_diary[i].dist;
            total += walk_diary[i].dist;
            printf("%7.2lf %7.2lf %8.2lf %-30s", walk_diary[i].dist,
                m_total, total, walk_diary[i].comment);
        }
        if((c < map_c) && (map[c+1].dist < total)) {
            while((c < map_c) && (map[c+1].dist < total)) { c++;}
            printf(" (%s到着", map[c].name);
            if(c < map_c) { printf(" %sまで%.2lfkm！",
                map[c+1].name, map[c+1].dist - total); }
            printf(")");
        }
        printf("¥n");
    }
    printf("%.2lfkm %sから%.2lfkm地点",
```

```
        total, map[c].name, total - map[c].dist);
    if(c < map_c) { printf(" %sまで%.2lfkm",
              map[c+1].name, map[c+1].dist - total); }
    else { printf(" ★★目標達成！★★ ¥n"); }
}
```

ヒント
*1：拡張子に注意して保存しましょう。

2 入力できたら、「diary.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、diary.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o diary diary.c
```

4 プログラムを実行する。なお、プログラムを実行する前に、「dat」という名前のディレクトリがあることを確認しておく

```
C:¥source>diary.exe -view
```

▼

```
【2009年4月のデータ】
41.20km 千葉から1.20km地点 仙台まで588.80km
       歩距離   月累計    全累計
 1 |    6.00     6.00     47.20 桜はまだだ
 2 |    2.00     8.00     49.20 途中で雨。断念
 3 |    7.30    15.30     56.50 暑い
 4 |    6.50    21.80     63.00 良い天気！
(略)
12 |    7.00    44.80     86.00
13 |    5.25    50.05     91.25 桜が散ってしまった
14 |
15 |    6.50    56.55     97.75 風が強かった
(略)
30 |
97.75km 千葉から57.75km地点 仙台まで532.25km
```

月の記録が表示されれば成功！

解説

1 今月までの歩行距離の累計を計算する

日記の表示プログラムでは、歩行距離の累計も表示します。ここでは、他のディレクトリにあるファイルデータを取得する方法について考えてみます。

(1) ディレクトリのファイル一覧の取得

データファイルは月ごとの記録なので、その月内の累計歩行距離はそのファイル内で計算できます。しかし、日記をつけはじめてからの累計歩行距離を計算するには、全ログファイル（これまでの記録データ）を参照する必要があります*2。

opendir関数を使って、datディレクトリ下のファイル一覧を取得します。

【8-3_sample1.c】

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>*3

int main() {
    DIR *dir;
    struct dirent *dp;

    dir=opendir("dat/");
    while((dp=readdir(dir)) != NULL) {
        printf("%s¥n", dp->d_name);
    }
    closedir(dir);
    return 0;
}
```

▼

```
.
..
walk200901
walk200902
```

datディレクトリ下のファイルとディレクトリの一覧が取得できました。「.」は自分自身のディレクトリを表し、「..」はひとつ上のディレクトリです。この2つは、必ずディレクトリの一覧に存在します。残りが純粋なdatディレクトリ以下のファイルおよびディレクトリです。

ディレクトリを見るには、opendir関数でオープンし、readdir関数で読み込みを行います。

ヒント

*2：月ごとの累計を別ファイルにまとめて記録しておく方法もありますが、今回はディレクトリ一覧取得の学習を行うため、毎回、全ファイルを参照するようにします。

ヒント

*3：一連の関数を使用するには、dirent.hファイルとsys/types.hファイルをインクルードします。

ヒント
*4：他のメンバも知りたい場合は、C:¥mingw-jp¥include¥dirent.hを参照してください。

readdir関数はdirent構造体へのポインタを返します。この構造体もC言語で用意されている構造体で、取得したファイルやディレクトリの情報が入ります。詳細はdirent.hファイルの中に書いてあります。今回はファイル名を扱うd_nameメンバだけ*4を利用します。

ディレクトリをオープンしたら、データファイル名のみをdatafilesに記録します。

```
char *datafiles[100];
int file_c = 0;

dir=opendir("dat/");
while((dp=readdir(dir)) != NULL) {
    if(strncmp(dp->d_name, "walk", 4) == 0) {
        datafiles[file_c] = (char*)malloc (strlen(dp->d_name) + 1);
        strcpy(datafiles[file_c++], dp->d_name);
    }
}
closedir(dir);
```

ヒント
*5：それぞれの関数を使うときは、必要なヘッダーファイルをインクルードします。この場合はstring.hとstdlib.hです。

strncmp関数*5はstrcmp関数の親戚で、第3引数に比較する文字列の長さを指定できます。

データファイル名は最初に「walk」という4文字がついているので、その場合はデータファイルであるとみなします。

メモリの確保には第6日で学習したmalloc関数*5を使って、データファイル名を記録します。

一覧をすべて読みおわったら、closedir関数でクローズします。ディレクトリのオープン・クローズはファイルのオープン・クローズと似ています。扱うのはファイルポインタではなく、DIR型のディレクトリポインタです。

(2) データファイルの参照

取得したデータファイルの一覧からファイルをひとつひとつ調べて*6、歩行距離の累計を求めます。

今月のデータファイルの内容は詳細に表示するため、あとで別にオープンします。今月以外のファイルを全て開いて、それぞれ次の処理を行います。

```
char str[128];
diary tmp;
double dist[31];

while(fgets(str, sizeof(str), fp) != NULL) {
    sscanf(str, "%d %lf %s¥n",  &tmp.day, &tmp.dist, tmp.comment);
    if(tmp.day > 0) { dist[tmp.day-1] = tmp.dist; }
}
```

データは1行ごとに「日付 歩行距離 コメント」の情報が記録されています。必要なのは歩行距離で、これをdouble型配列distに保存*7します。

最後に、配列distの各要素をプラスすれば、歩行距離の累計が算出されます。

ひとつのファイルがおわったら、次のファイルも同様にします。double型配列distの全要素値を0でクリアして、同じように月の歩行距離を記録し、最後に累計を出します。これを全てのファイルで行えば、今月以外の歩行距離の累計が算出されます。

walk200811	walk200812		walk200903	
1 9.00 寒い	1 4.00		1 10.00	
2 4.50	2 8.20 疲れた	……	2 12.20	
3 5.00	4 4.40		3 4.50	
：	：		：	
累計 200km ＋	250km	＋……＋	180km	＝累計 ○○○km

C O L U M N

日付のエラーチェック

今回のプログラムでは、もし同じ日付のデータがあれば上書きされますが、2月30日などのように間違った日付を入れたデータについては、手動で削除しない限り、有効なままです。うるう年についても考慮していません。

また、2100年など、今月よりも先の日付のデータも作成され、読み込まれてしまいます。

今回のプログラムではそれらのチェックを行っていませんが、余裕のある人は、これらの対策を自力で行ってみてください。データを読み込むとき、またはデータを作成するときに、チェック機能をつけるのが一般的です。

2 構造体配列

日記の1日分のデータは、「日」「距離」「コメント」の3種類です。この3種類は構造体を利用してひとつのデータ型、ここではdiary型として扱っています。この構造体を配列にして最大31個用意すれば、ひと月分のデータを格納できます。

```
diary walk_diary[31];
```

walk_diary[0]には1日のデータ、walk_diary[1]には2日のデータが入ります。

ヒント

*6：opendir関数でデータファイルの一覧を取得しているときに、それぞれファイルをオープンして中身を参照する方法もあります。今回は、一度データファイル一覧を記録し、ディレクトリをクローズしてから、改めてファイルをひとつひとつオープンして中身を参照します。

ヒント

*7：「日付-1」の添え字の場所に保存されるので、もし同じ日付のデータがあれば上書きされます。これは、2時限目に作ったデータファイル記録機能の仕様に沿っています。

●構造体diaryと構造体配列walk_diaryの関係

構造体

int day;

double dist;

diary型

char[32] comment;

diary walk_diary[31]

構造体配列

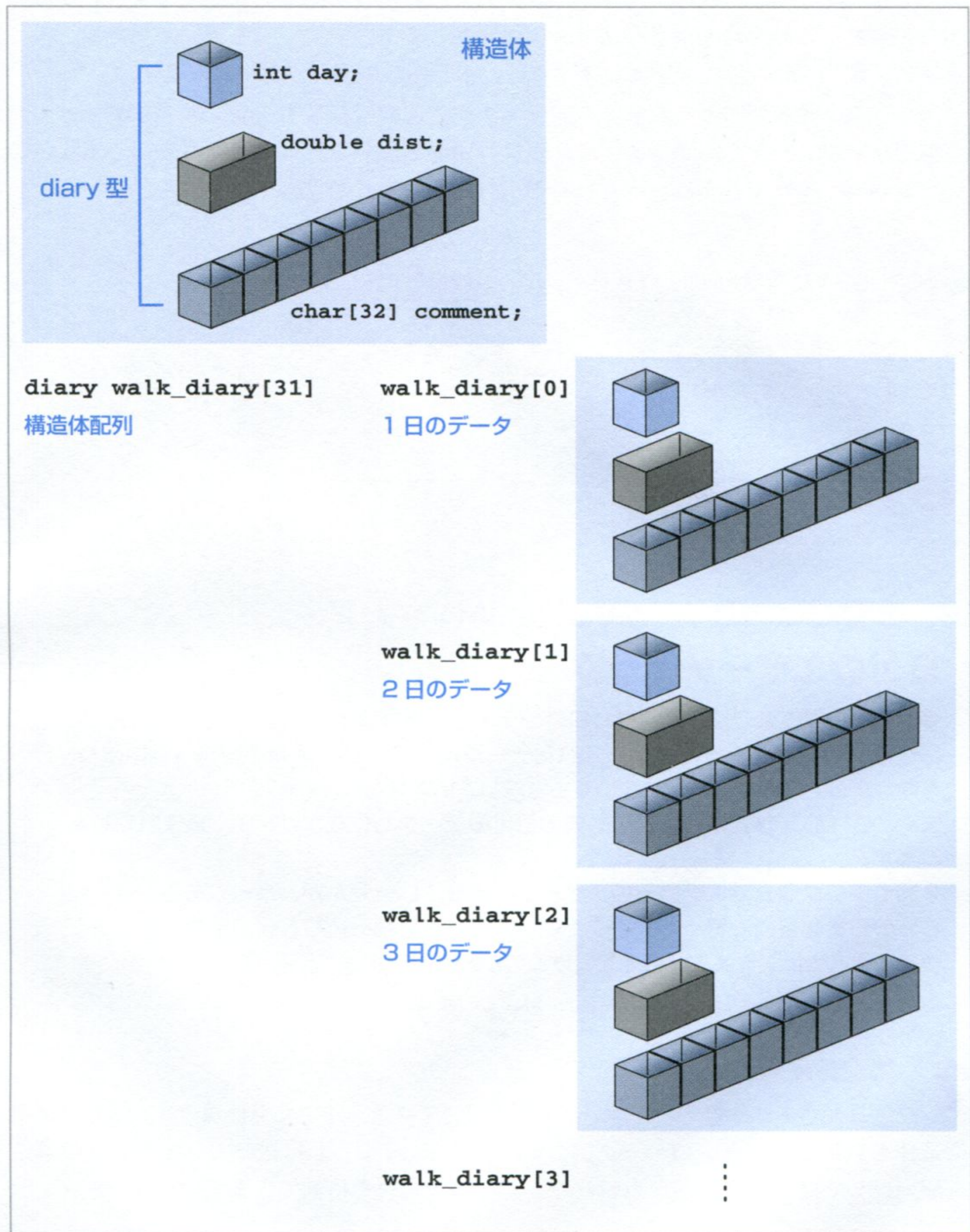
walk_diary[0]

1日のデータ

walk_diary[1]

2日のデータ

walk_diary[2]

3日のデータ

walk_diary[3]

構造体配列の初期化は、3種類のデータを{ }で括り、それぞれをカンマ「,」で区切って定義します。

メンバの参照には添え字を使います。例えば、walk_diaryの4日のコメントは、

```
walk_diary[3].comment
```

構造体型配列の変数名[添え字].構成データのメンバ名

ヒント

*8：コメントは、ない（記録していない）場合があります。もし同じ日付のデータがある場合はあとから記録したものが上書きされますが、コメントがない場合は、前のデータが残ったままなので、最初にクリアします。

で参照できます。構造体型配列の変数名に続いて、何番目のデータを参照したいか、添え字番号を指定します。あとは構造体型データの参照方法と同じです。

では、今月のデータファイルから、日記のデータを取り出してみましょう。読み込みを行う部分は次のとおりです。

```
diary tmp;
diary walk_diary[31];

while(fgets(str, sizeof(str), fp) != NULL) {
    strcpy(tmp.comment, "");*8
    sscanf(str, "%d %lf %s¥n",  &tmp.day, &tmp.dist, tmp.comment);
    if(tmp.day > 0) {
        walk_diary[tmp.day-1].day = tmp.day;
        walk_diary[tmp.day-1].dist = tmp.dist;
        strcpy(walk_diary[tmp.day-1].comment, tmp.comment);
    }
}
```

一度、diary型の変数tmpにそれぞれ読み込んで、そこから配列に格納します。変数commentのみ文字列なので、strcpy関数を使用しましょう。

あとは構造体配列walk_diaryを、1日から順に表示するだけです。すべて31日までの表示ではなく、それぞれの月の最終日を考えて表示しましょう*9。

ヒント
*9：今月データの表示については、うるう年の場合も考慮に入れています。

ヒント
*10：東京からの距離データはおおよその値です。仙台→札幌は、北海道を一周してきた距離で計算しています。

3 日本一周データを使用する

ただ歩行距離を出しただけではつまらないので、日本一周するとしたら、現在どのあたりまで歩いたのかを表示しましょう。バーチャル日本一周ウォーキングです。

東京を出発し、東北、北海道、北陸、山陰、九州、関西、東海、東京というルートにします。おおざっぱですが、データは次のものを用意します*10。

```
typedef struct {
    char name[10];
    double dist;
} dist_data;

dist_data map[] = {
        { "東京", 0.0 },{ "千葉", 40.0 },{ "仙台", 630.0 },
        { "札幌", 2800.0 },{ "金沢", 4100.0 }, { "博多", 5000.0 },
        { "広島", 6330.0 },{ "名古屋", 8000.0 }, { "東京", 8300.0 }};
```

このデータを利用して、累計歩行距離から現在地点を割り出しましょう。

まず、たとえば4月のデータを表示する場合、3月までの歩行距離の累計と、それはどの地点であるか、次の地点までどのぐらいであるかを表示します。

```
 51.20km 千葉から 11.20km 地点  仙台まで 578.80km
```

そして、今月の日記一覧では、どこかの記録地点に到達した時点でそれを表示します[*11]。

```
       歩距離   月累計    全累計
28 |   20.00   300.25    656.45 今日は寒い（仙台到着 札幌まで 2143.55km！）
```

最後にもう一度、今月のデータを含めた歩行距離の累計と、それはどの地点であるか、次の地点までどのぐらいであるかを表示します。

もし最後の地点までたどり着いたら、「目標達成！」と表示します。

ヒント

*11：表示が途中で折り返してしまう場合は、コマンドプロンプトのプロパティを開いて、「レイアウト」タブの「画面バッファのサイズ」の幅を変更しましょう。

まとめ

ディレクトリの一覧取得と、構造体配列について学習しました。ファイルの扱いや構造体、および配列など、今まで学習してきたことの応用になっています。理解がいまひとつ……という人は、もう一度それらの項目を復習してみてください。

練習問題

Q

距離データdist_data mapを、付属CD-ROMに収録したファイル「day08-03¥map.dat」から取得するように、diary.cの自作関数viewDataを変更しなさい。
ただし、map.datはC:¥sourceディレクトリにコピーして実行することを前提とする。

解答は巻末に

第 9 日

25ゲームを作ろう

1時限目　マクロを理解しよう

2時限目　25ゲームのしくみを考えよう

3時限目　25ゲームを完成させよう

本日は25ゲームを作成します。
25ゲームとは5×5のマス目に1～24の数値をランダムに配置し、それを順番になるように入れ替えるゲームですが、3×3で1～8のゲームにも対応できるようにします。どちらにも対応できるようにプログラムを作成するには、マクロという機能を使って変更箇所が1箇所ですむようにします。

今日作るプログラムについて

25ゲームゲーム

25ゲームでは、5×5のマスを作り、1～24までの数値をひとつのマスにひとつずつ、適当な位置に割り振ります。5×5では25個のマスができるので、1～24までの数値を配置すると、1箇所だけ空きます。その空いた場所に、隣接するマスに入っている数値を移動します。

数値をどんどん移動して、数値が1～24まで順に並んだ状態にすれば完成です。

スタート

9	21	7	3	4
12	11	16	10	19
14	8	13	2	17
23	5	24	20	15
18	22	6	1	

移動

9	21	7	3	4
12	11	16	10	19
14	8	13	2	17
23	5	24	20	
18	22	6	1	15

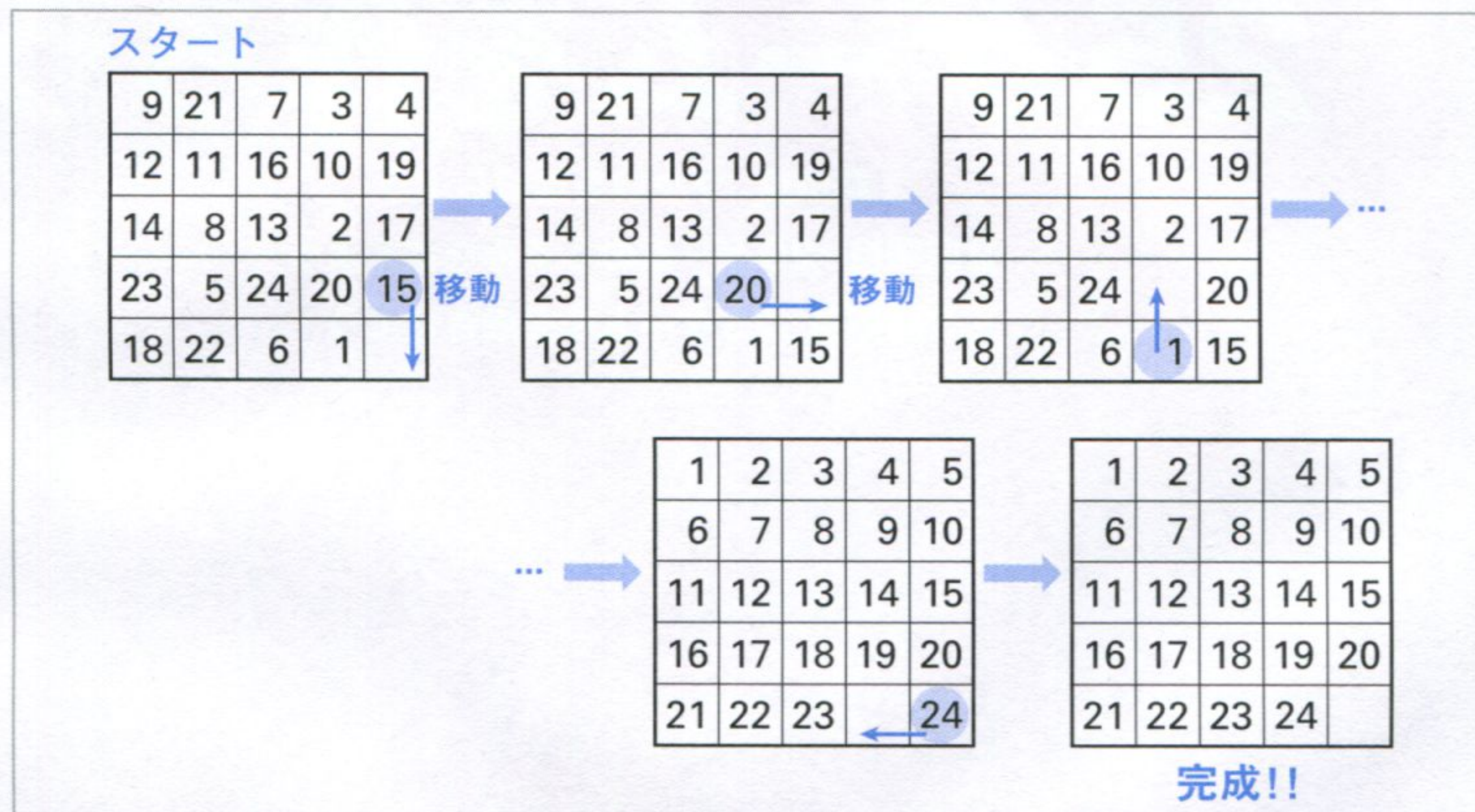
移動

9	21	7	3	4
12	11	16	10	19
14	8	13	2	17
23	5	24		20
18	22	6	1	15

…

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23		24

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	

完成!!

25ゲームの実際の動作

1 25ゲームを実行すると、ゲームタイトルと1～24の数値がランダムに並んで表示される

```
C:¥source>25game.exe
```

▼

```
【 25 ゲーム】(0 を入力すると途中で終了)
10 02 13 23 06
14 08 19 01 15
03 04 05 16 11
09 21 17 20 07
18 24 12 22
動かす数字を入力して下さい >
```

2 空白部分に移動する数値を入力して、[Enter] キーを押す

```
動かす数字を入力して下さい > 22
```

3 数値が移動し、入力した数値部分が空白になる。なお、移動できないマスの数値が入力された場合は移動を行わない

```
動かす数字を入力して下さい > 22
 10 02 13 23 06
 14 08 19 01 15
 03 04 05 16 11
 09 21 17 20 07
 18 24 12    22
動かす数字を入力して下さい >
```

4 上記②～③を、すべての数値が１～24まで左上から右下のマスまで順に揃うまで繰り返す

5 揃ったら、「成功！！」という文字列と移動回数が表示され、ゲーム終了

```
動かす数字を入力して下さい > 24
 01 02 03 04 05
 06 07 08 09 10
 11 12 13 14 15
 16 17 18 19 20
 21 22 23 24

☆☆☆成功！！☆☆☆
移動した回数は 102 回でした
```

途中でやめたい場合は０を入力する

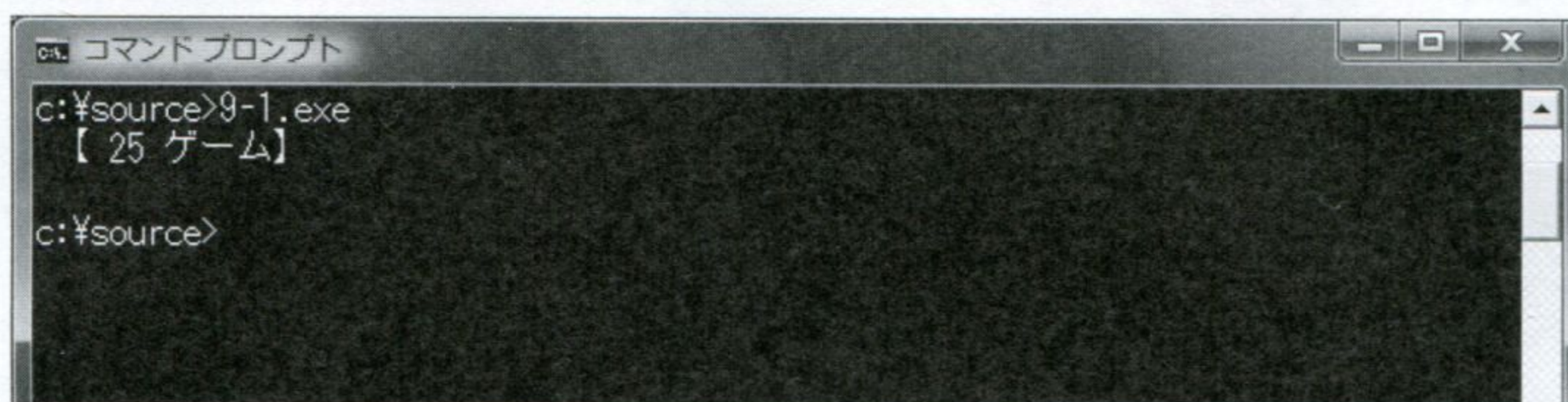
```
動かす数字を入力して下さい > 0


======== 終了 ========
```

この時限ではマクロの基礎を学習します。

今回作成する例題

```
C:¥source>9-1.exe
 【 9 ゲーム】

C:¥source>
```

ゲームのマス数を表示する

```
c:¥source>9-1.exe
 【 25 ゲーム】

c:¥source>
```

サンプルファイルはこちら 10days_c ▶ day09-01 ▶ 9-1.c

●このレッスンのねらい

「マクロ」という言葉を聞いたことのある人もいると思います。表計算ソフトの定番「Excel」で操作や計算などを自動で行ってくれる機能をマクロといいますが、C言語でもマクロ機能が用意されています。値を置き換えたり小さい計算を行ったりすることを、マクロ処理と呼びます。
本レッスンでは、値の置き換えや計算を行うマクロについて学習しましょう。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>

#define MTX 3
#define CMTX(x) (x)*(x)

int main() {
    printf("【%d ゲーム】¥n", CMTX(MTX));

    return 0;
}
```

2 入力できたら、「9-1.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

ヒント
*1：拡張子に注意して保存しましょう。

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、9-1.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 9-1 9-1.c
```

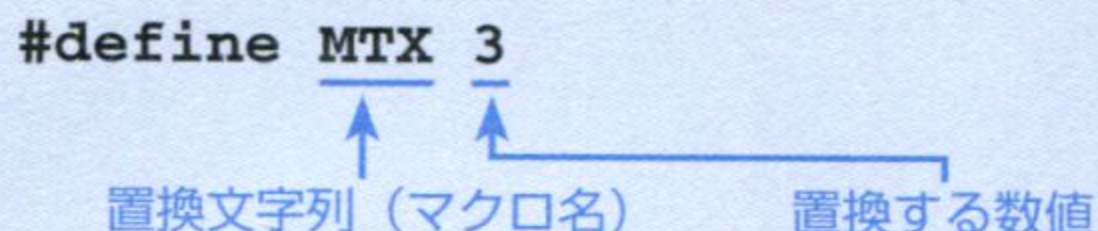
4 プログラムを実行する。9-1.cのMTXの値が3なら【9 ゲーム】、5なら【25 ゲーム】と表示されれば成功！

```
C:¥source>9-1.exe
```

▼

```
【9 ゲーム】
```

解説

1 置換を行うマクロ

まずは、文字列を値に置き換えるマクロを使ってみましょう。

(1) マクロの書き方

使用するマスの数は5×5ですが、完成するまでの確認は3×3で行います。マスの行列数を表す文字列「MTX」を、数値の3に置き換えるマクロを作ります。書き方は次のとおりです。

【置換マクロの書き方】

```
#define MTX 3
```

置換文字列（マクロ名）　置換する数値

このように定義すると、文字列MTXが数値3に置き換えられます[*2]。マクロ名は自由につけられますが、すべて大文字にしておくと、コードが見やすくなります。

マクロの定義は、プログラムの最初にまとめて書きます。置換を行うマクロ文は、include文のあとに書きましょう。

ヒント
*2：マクロ文の最後にセミコロンはつきません。注意しましょう。

では、25ゲームのゲーム名を表示するだけのプログラムを、マクロを使って作ってみます。マスの行列数を表す文字列MTXに数値3をマクロで定義します。

基本的に、置換マクロは変数の使い方と同じです。

```
printf("【%d ゲーム】¥n", MTX*MTX);
```

プログラム中の文字列「MTX」は数値3に置き換わるので、もしマスの行列数を変更して5×5のゲームを作るときは、MTXの数値だけを変更します。

```
#define MTX 5
```

また、マクロは他のデータ型の値でも使うことができます。

```
#define STR "Hello World!"
```

文字列の場合はダブルクォート、文字の場合はシングルクォートで括りましょう。

(2) マクロを使う利点

次のプログラムのように、MTXの値はただの変数を使って書くこともできます。

```
#include <stdio.h>

int MTX = 3;

int main() {
    printf("【%d ゲーム】¥n", MTX*MTX);
    return 0;
}
```

これでも9-1.cと同じ結果になります。マスの数を変更するには、int型変数MTXの初期値を変更します。

ではマクロを使う利点はなんでしょうか？

プログラムの中で、変数MTXの値と同じ数値を、別の配列変数の大きさに指定したい場合、

```
int MTX = 3;

int main() {
    int d[MTX];
    (略)
```

と変数MTXを利用したいところですが、これはコンパイラによってはエラーになります。d[MTX]と書いても、「配列の大きさには定数を指定してくれ」とコンパイラに怒られてしまいます。このため、この方式では、

```
int MTX = 3;

int main() {
    int d[3];
    (略)
```

と定義し、数を変えたいときは、変数MTXの初期値と配列dの大きさの両方を変更しなければなりません。マクロを利用すると、この手間がなくなります。

```
#define MTX 3

int main() {
    int d[MTX];
    (略)
```

マクロMTXを配列変数の大きさに指定しても、コンパイルエラーにはなりません。これは、マクロがコンパイルを行う前に処理される機能だからです。マクロはプリプロセッ

サと呼ばれる、コンパイルの前に行う準備処理で展開されます。置換を行う#defineマクロは、マクロ文字列をすべて対応する値に置き換えます。

●マクロはプリプロセッサで処理される

```
#include <stdio.h>

#define MTX 3

int main() {
    int d[MTX];

    printf("【 %d ゲーム】¥n", MTX*MTX);
      :
    return 0;
}
```

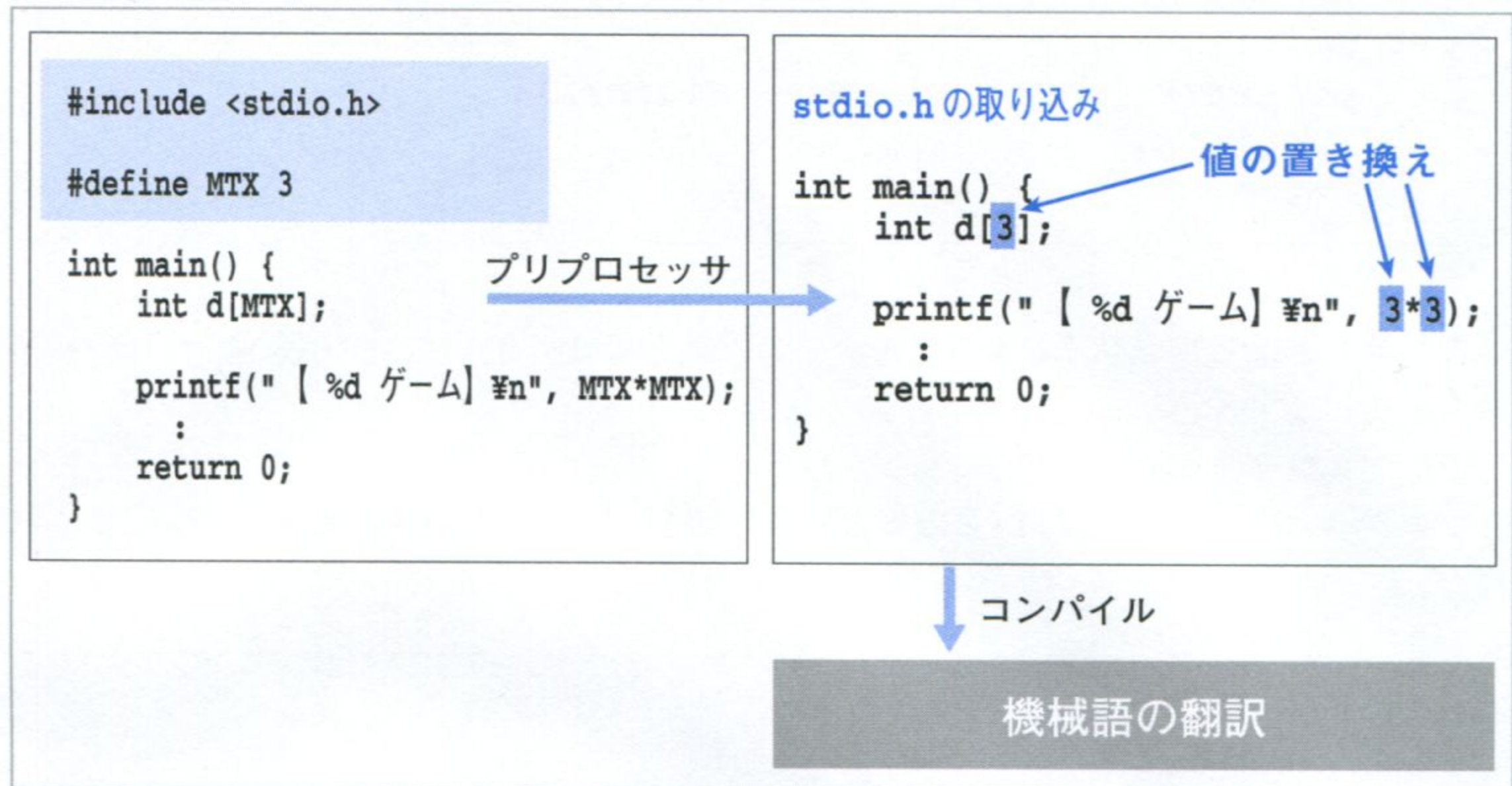
プリプロセッサ →

stdio.h の取り込み

値の置き換え

```
int main() {
    int d[3];

    printf("【 %d ゲーム】¥n", 3*3);
      :
    return 0;
}
```

↓ コンパイル

機械語の翻訳

つまり、コンパイルの前の段階のプリプロセッサ[*3]で、

```
int main() {
    int d[3];
```

に置き換えられているので、エラーが発生しないのです。

このように、マクロを使えばプログラム中で同じ数値を何度も定義する必要がなくなり、値を簡単に変えることができます。

ヒント

*3：プリプロセッサについては第10日にも説明します。

(3) マクロを書く位置

マクロは、include文のあとに書きます。

```
#include <stdio.h>
#include <stdlib.h>

// マクロ定義
#define MTX 3

// グローバル変数と関数のプロトタイプ宣言

int main() {
    (略)
    return 0;
}
```

マクロは、実はプログラム中のどこに書いてもかまいません。マクロ定義の次の行から、その定義が有効になります*4。

```
int main() {
    #define MTX 3
    printf("【 %d ゲーム】¥n", MTX*MTX);
```

次のように書くと、1行目の時点ではMTXの値がわからないので、コンパイルエラーになります。

```
    printf("MTXの値：%d¥n", MTX);
    #define MTX 3
    printf("【 %d ゲーム】¥n", MTX*MTX);
```

マクロはプログラム全体で使うことが多いうえ、プログラムの見やすさを考慮すると、最初にまとめて定義したほうがよいでしょう。

ヒント

*4：#defineの定義を無効にするには#undefを使います。

COLUMN

「真」「偽」と「TRUE」「FALSE」

マクロの定義で一般的によく使われるのが「真」「偽」です。C言語では偽を0で、真をそれ以外（主に1）で表現します。これをマクロでTRUE、FALSEという文字列で設定しておきます。

```
#define TRUE  1
#define FALSE 0
```

すると、プログラムは次のように書き直すことができ、他の人が見てもわかりやすいプログラムになります。

```
if(kekka == 0) {
      ↓
if(kekka == FALSE) {
```

フラグのONを1、OFFを0などと定義しておくとわかりやすくなります。今後はマクロをどんどん利用しましょう。

② 引数つきマクロ

マクロは文字列の置き換えだけでなく、関数のように使う方法もあります。例えば、25ゲームの最初に表示する文字列は、printf関数を使って表示しています。

```
printf("【 %d ゲーム】¥n", MTX*MTX);
```

このとき表示する数値は「MTX*MTX」で計算しています。この計算式をマクロ化して、

```
printf("【 %d ゲーム】¥n", マクロ名(MTX));
```

と指定するプログラムを作ってみましょう。マクロ名のあとに( )をつけ、その中に引数MTXを指定します。返ってくる値は、「MTX*MTX」の計算結果です。引数を指定してそれに伴った値を返してもらう……まさに関数のように使います。

引数つきマクロの書き方[*5]は次のとおりです。

ヒント
*5：マクロは基本的に1行で書きますが、定義が長くなるときは行末に「¥」をつけると、次の行に続きを書くことができます。

【引数つきマクロの書き方】

```
#define CMTX(x) (x)*(x)
```

マクロ名 → CMTX(x)
計算式 → (x)*(x)

引数xに入ってきた値を、計算式(x)*(x)にあてはめる計算式が定義されました。引数に定義する変数はローカル変数のような扱いなので、好きな変数名を使います[*6]。関数の定義とは違い、引数に型を宣言する必要がないのが利点です。

ヒント
*6：引数つきマクロ名も、置換を行うマクロ名と同様、好きにつけることができます。ただし、こちらもできるだけ、すべて大文字の名前をつけましょう。

③ 演算子の優先順位

さて、ここで少し考えてみましょう。引数つきマクロを、次のように定義しなかったのはなぜでしょうか。

```
#define CMTX(x) x*x
```

このプログラムでは、引数に5や3など、ひとつの数値を指定しているので何も問題ありませんが、引数に計算式を指定した場合を考えてみましょう。

```
CMTX(5-2)
```

この場合、「x*x」の結果は「5-2*5-2」になります。これは「3*3」にならず「5-10-2」、つまり、「-7」になってしまいます[*7]。

ヒント
*7：9-1.cを一時変更して、実際に試してみましょう。

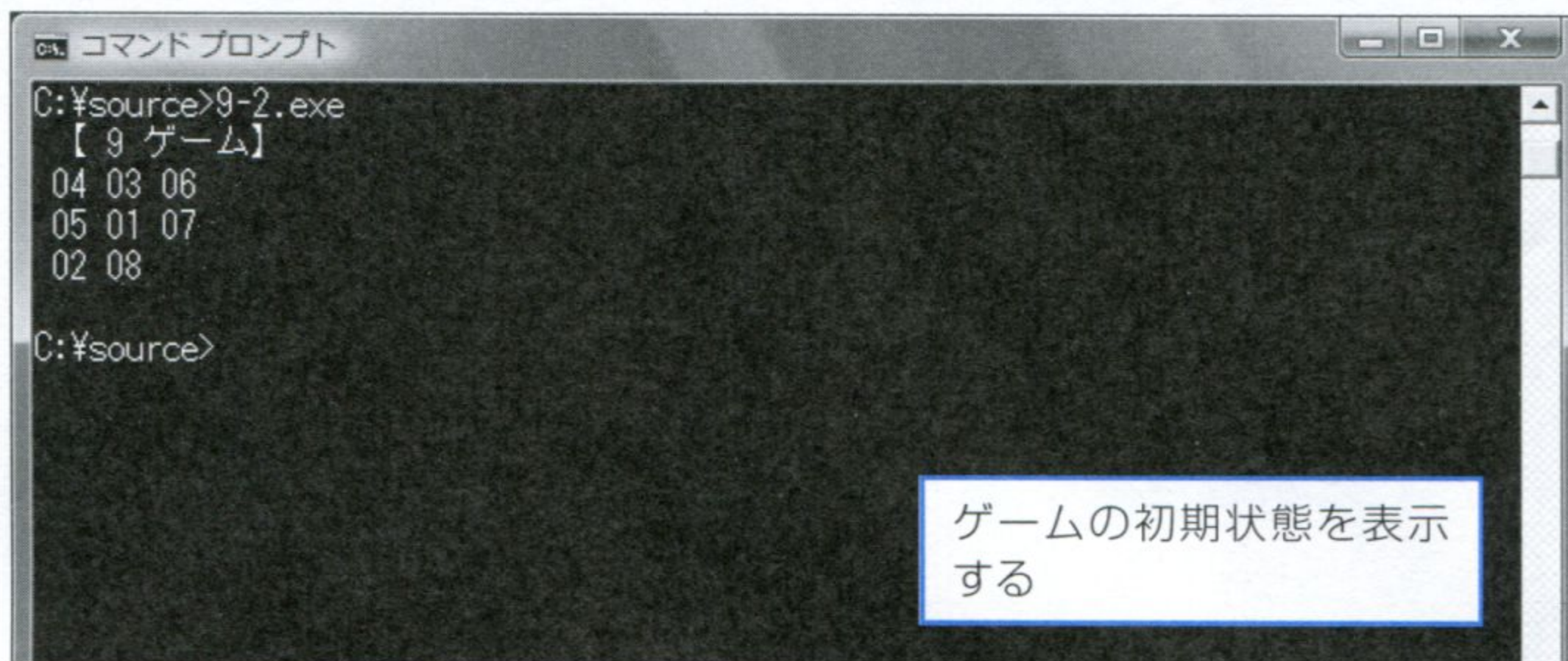
これは、演算子「-」よりも演算子「*」の方が、優先順位が高いためです。一般的な算数の四則演算で、「＋」「－」よりも「×」「÷」の優先順位が高いという法則と同じです。

C言語での演算子の優先順位は、次のようになっています。

●演算子の優先順位

順位	演算子
1	( ) [ ] -> . 後置++ 後置--
2	! ~ 前置++ 前置-- 間接* アドレス& sizeof
3	キャスト（型）
4	乗算* / %
5	+ -
6	<< >>
7	< <= > >=
8	== !=
9	ビット&
10	^
11	|
12	&&
13	||
14	?:
15	= += -= *= /= %= &= |= ^= <<= >>=
16	,

このように優先順位を考慮して、引数にどんな値がきてもよいように、(x)*(x) と、引数部分を ( ) で括ります。これはどんな複雑なマクロでも同じなので、基本的にマクロの引数値は、それぞれ ( ) で括って宣言しましょう。

まとめ

この時限では、マクロについて学習しました。

今までに作成したプログラムを見直したとき、おそらく「ここをマクロに直したい」と思う箇所がいくつも出てくると思います。練習になるので、時間のあるときにいくつか試してみてください。マクロ使用前と実行結果が同じであれば成功です。

25ゲームのしくみを考えよう

25ゲームの作り方を考え、ここでは25ゲームのマスを初期状態にするところまで作ります。

今回作成する例題

```
C:¥source>9-2.exe
 【 9 ゲーム】
 04 03 06
 05 01 07
 02 08

C:¥source>
```

ゲームの初期状態を表示する

サンプルファイルはこちら 10days_c ▶ day09-02 ▶ 9-2.c

●このレッスンのねらい

このゲームプログラムは、今までのプログラムに比べると、少々長いプログラムになります。このため、機能をわかりやすく分割し、それぞれを関数化します。先に関数の仕様だけ決めて、あとから中身を追加していく方法でプログラムを完成させましょう。
この時間では、まずmain関数とゲームのマスを初期化する関数のみを作成します。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define MTX 3
#define CMTX(x) (x)*(x)

int board[MTX][MTX];// ボード

void initBoard(void);

int main() {
    srand(time(NULL));
    printf("【%d ゲーム】¥n", CMTX(MTX));
    initBoard();

    return 0;
}

// ボードの初期化を行う関数
void initBoard(void) {
    int i, j;
    int m, nokori = CMTX(MTX)-1;
    int r; // ランダムな値
    int digit[CMTX(MTX)-1]; // 残り表示数値を記録する

    for(i = 0; i < nokori; i++) {
        digit[i] = i+1;
    }

    for(j = 1; j <= MTX; j++) { // 縦方向の繰り返し
        for(i = 1; i <= MTX; i++, nokori--) { // 横方向の繰り返し
            if((j == MTX) && (i == MTX)) { break; }
            if(nokori > 1) { r = rand()%nokori; }
            else { r = 0; }      //nokori の数が 1 のときは digit[0]
            board[j-1][i-1] = digit[r]; // ボードに代入する
            printf(" %02d", digit[r]);
            for(m = r; m < nokori-1; m++) { digit[m] = digit[m+1]; }
```

```
        }
        printf("¥n");
    }
    board[MTX-1][MTX-1] = 0; // 最後のマスに0を代入する
}
```

ヒント
＊1：拡張子に注意して保存しましょう。

2 入力できたら、「9-2.c」という名前[＊1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、9-2.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 9-2 9-2.c
```

4 プログラムを実行する。1～8までの数値がランダムに並べられていたら成功！

```
C:¥source>9-2.exe
```

▼

```
【 9 ゲーム】
05 07 04
06 03 01
02 08
```

解説

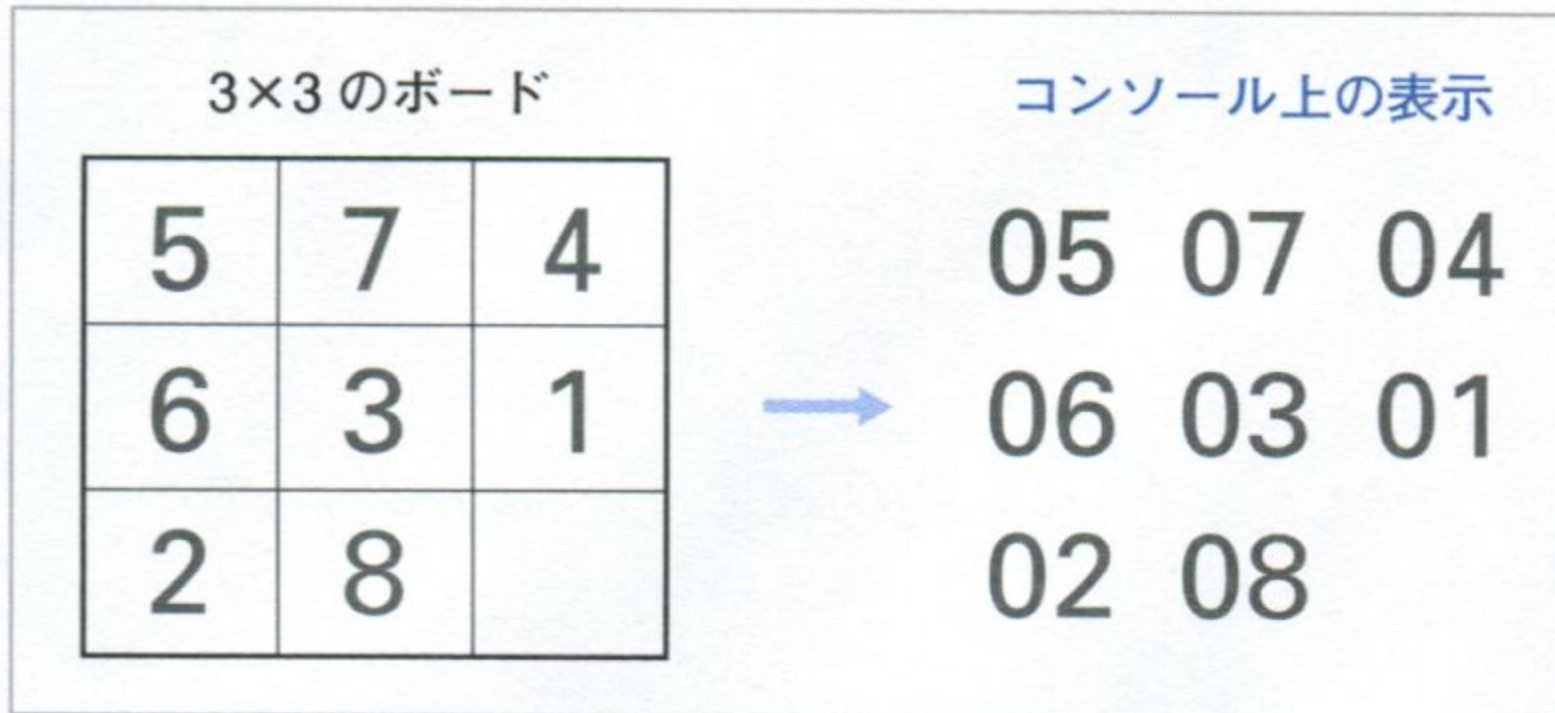
1 25ゲームのしくみを考える

25ゲームは、ランダムに配置された数値を空いているマスを利用して移動させ、1～24まで順に並べるゲームです。

プログラムが完成するまでは、動作確認を含めて3×3の9ゲームとして作成しましょう。9ゲームの場合は、1～8までを順に並べることになります。

このゲームで基本になるのは、3×3の数値を並べた状態のものです。これを、ゲームのボードと呼びましょう。

●9ゲームのボード

3×3のボード				コンソール上の表示
5	7	4	→	05 07 04
6	3	1		06 03 01
2	8			02 08

ゲームの流れは次のようになります。

●ゲームの流れ

・ゲームのボードの初期化
・移動する数値の入力と完成したかどうかのチェック（繰り返し作業）
・最後まで揃えることができたか、失敗（リタイア）したかを表示

これらすべてをmain関数の中で書くのは大変です。よって、次の3つを関数化しましょう。

●ゲームの中で関数化する処理

[自作関数]	[担当する処理]
initBoard	ボードの初期化
moveDigit	数値の移動
writeBoard	ボードの表示と完成かどうかのチェック

ゲームは、移動する数値の入力と完成したかどうかのチェックを、繰り返し行います。
繰り返しを終了するには、数値がすべて順番に並ぶか（成功）、0を入力して途中でリタイアするか（失敗）のどちらかです。
この繰り返し処理はmain関数の中で行います。いつ終了するかわからない繰り返しなので、無限にループするwhile(1)を使いましょう。

●ゲームの流れと自作関数

```
ボードの初期化; (initBoard)
while(1) {
        動かす数値を入力する (0を入力したら繰り返し終了);

        数値を移動させる; (moveDigit)
        ボードを表示し、同時に完成したかどうかチェックする; (writeBoard)
        ボードが完成していたら繰り返し終了;
}
結果の表示;
```

25ゲームは、あわせて3つの自作関数から構成されることになります。

自作関数moveDigitとwriteBoardは次の時限で作成するので、この時限はゲームのボードを初期化する自作関数initBoardを作成しましょう。

2 ゲームのボードを用意する

ゲームのボードを作成するには、2次元配列を使います。横方向をx、縦方向をyとします。

●25ゲームのイメージ

→ x方向 / ↓ y方向

	1	2	3	4	5
1	9	21	7	3	4
2	12	11	16	10	19
3	14	8	13	2	17
4	23	5	24	20	15
5	18	22	6	1	0

注）この図では5×5だが、最初は3×3の2次元配列を作る

xとyの数にはマクロを利用し、ゲームのボードをグローバル変数として定義しておきます。各マスには1～(MTX*MTX-1)までの数値が入るので、int型で定義します。空となるマスには、数値の0を入れておきます。

```
int board[MTX][MTX];// ボードの定義
```

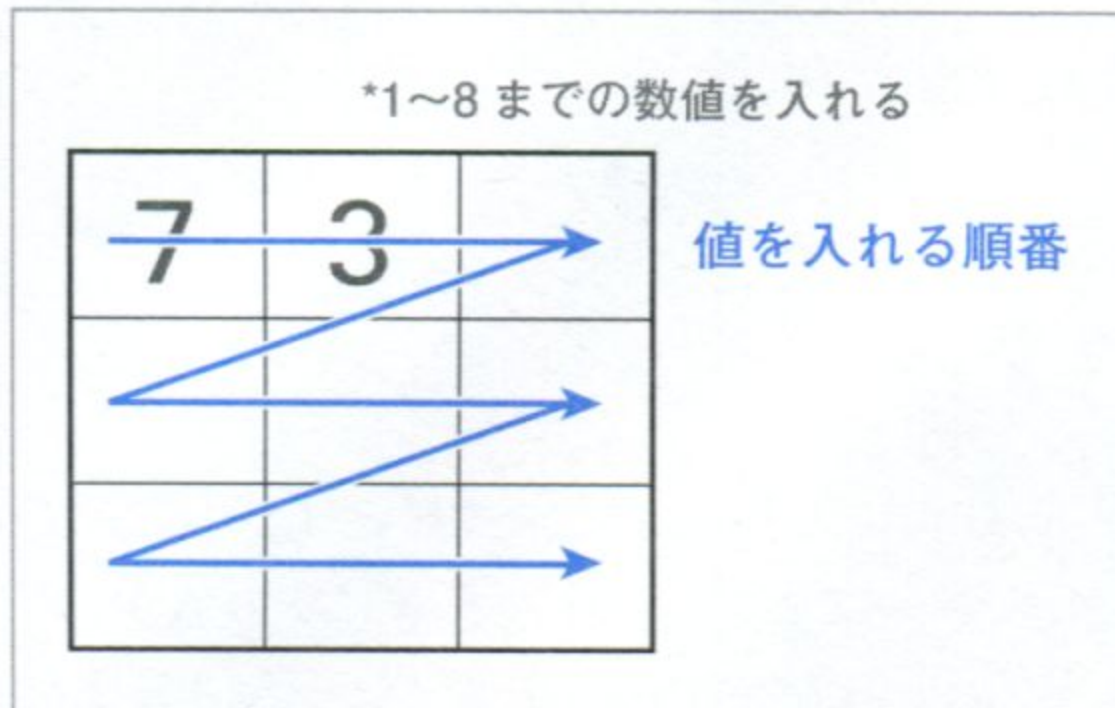
3 ゲームのボードを初期化する自作関数initBoard

ゲームのボードを初期化、つまり、1〜8までの数値をそれぞれゲームのボードを表す2次元配列に ランダムに配置します。最後のマスには、「空き」を意味する0を入れておきます。数値はボードの端から順に入れていきます。

●ボードに値を入れる順番

*1〜8までの数値を入れる

7	3	

値を入れる順番

ボードに配置する1〜8の数値はランダムにしたいので、rand関数を使います。しかし、すべてのマスに異なる数値を割り振らなければならないので、一度出てしまった数値はもう使えません。

第4日の脳トレゲームを作ったときの方法を使って、数値が重複しないようにしましょう。

```
int digit[CMTX(MTX)-1]; //残り表示数値を記録する
```

この配列に、前から順に1、2、3……、8までを代入します。ここに入っている数値は、表示することのできる数値です。

この中からランダムにひとつ選んでは、配列から減らしていきます。実際に配列の長さを短くするのは手間なので、選んだ数値を抜かして、その場所以降の配列を前に詰め、選択対象になる配列の範囲を「0〜7」→「0〜6」→「0〜5」……と減らしていきます*2。

ヒント

*2：ブラックジャックゲームで作ったように、配列を利用して「一度出た／出ない」を記録して、一度も出ない数値が選ばれるまで繰り返す方法もあります。

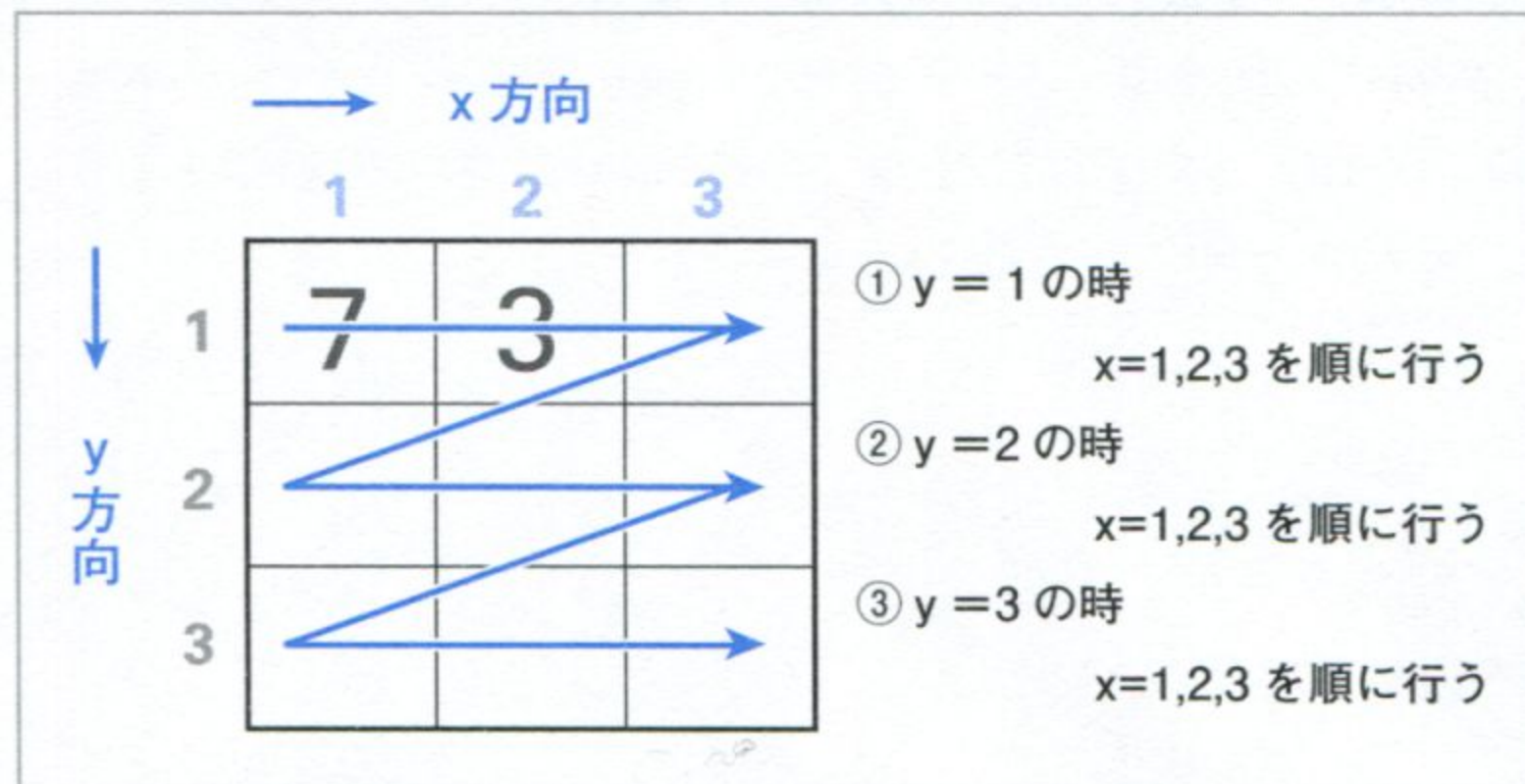
●ボードに表示する数値の選び方

配列の中身（）の部分は選択対象外

12345678 → rand()%8の値が4のとき→digit[4]の値5を表示
↓ digit[4]以降の配列をひとつずつ前につめる
1234678(8) → rand()%7の値が2のとき→digit[2]の値3を表示
↓ digit[2]以降の配列をひとつずつ前につめる
124678(88)
（続く）

しくみをきめたところで、実際にボードに数値を入れてみましょう。

ゲームのボードを表す2次元配列board[MTX][MTX]には、端から順にランダムな数値を格納します。この「端から順」を実現するためには、2次元配列独特の方法を使います。

●ボードへの数値の入れ方

「x=1, y=1」から値を入れます。次は、「x=2, y=1」に入れます。「x=3, y=1」の次はひとつ下の段に移動して、「x=1, y=2」に値を入れます。つまり、y=1のときにxの値が1～MTXまでのマスについて処理を行い、次に、y=2のときに同じくxの値が1～MTXまでのマスについての処理を行います。これを、y==MTXまで、すべての段で繰り返します。よって、

```
for(yの全ての段を1～MTXまで行う) {
        for(xのマス1～MTXまで行う) {
                数値決定・代入
        }
}
```

という二重のfor文を作ります。2つのfor文では、インクリメントする変数を別々に使います。よく使われるのは、yの方向の処理にj、xの方向の処理にiを使う方法です。

また、最後のマスである（j == MTX）&&（i == MTX）のときには、必ず「空き」を表す0を入れるので、ランダムな値を代入せずに繰り返しを終了します。

```
int nokori = CMTX(MTX)-1;    //残り表示数値の数

for(j = 1; j <= MTX; j++) {
    for(i = 1; i <= MTX; i++, nokori--) {
        if((j == MTX) && (i == MTX)) { break; }
            数値決定・代入；
            数値を出力する；
            配列digitの値をつめる；
    }
    改行を出力する；
}
```

ボードに数値を代入するのと同時に、コンソール上に表示します。yの値が変更された

ら次の段に移るので、改行を出力します。こうすることで、ボードらしく形を整えて表示できます。

なお、5×5では数値が1桁のものと2桁のものが出てくるので、必ず2桁の数値に直して表示しましょう。これもボードを整えて表示するための処理です。

```
printf(" %02d", 数値 );
```

これで1桁の数値の場合は、前に0がつきます。「1」は「01」となります。注意しなければならないのは、インクリメント用の変数iとjが、今回は両方とも1からはじまり、MTXまで繰り返し処理を行っている点です。配列の添え字は0からはじまるので、2次元配列boardや配列useDigitに入れる値は、iやj、それぞれから-1した場所に格納します。

プログラムを実行して、ダブリのない数値がランダムに表示*3されることを確認してください。

ヒント

*3：最後のマス、board[MTX-1][MTX-1]には数値0を入れますが、表示はしません。

まとめ

この時限では、ボードの初期化を行いました。次の3時限目で25ゲームを完成させましょう。

練習問題

Q

自作関数initBoardを改造して、初期化したボードがすでに揃ってしまっている場合の処理を追加しなさい。

[ヒント]
表示数値を格納する変数tmp_board[CMTX(MTX)]を作り、そこに表示予定の数を格納していく。tmp_boardの数値が1からCMTX(MTX)まで順に並んでいたら、もう一度並べ直す。

初期化したボード

5	7	4
6	3	1
2	8	

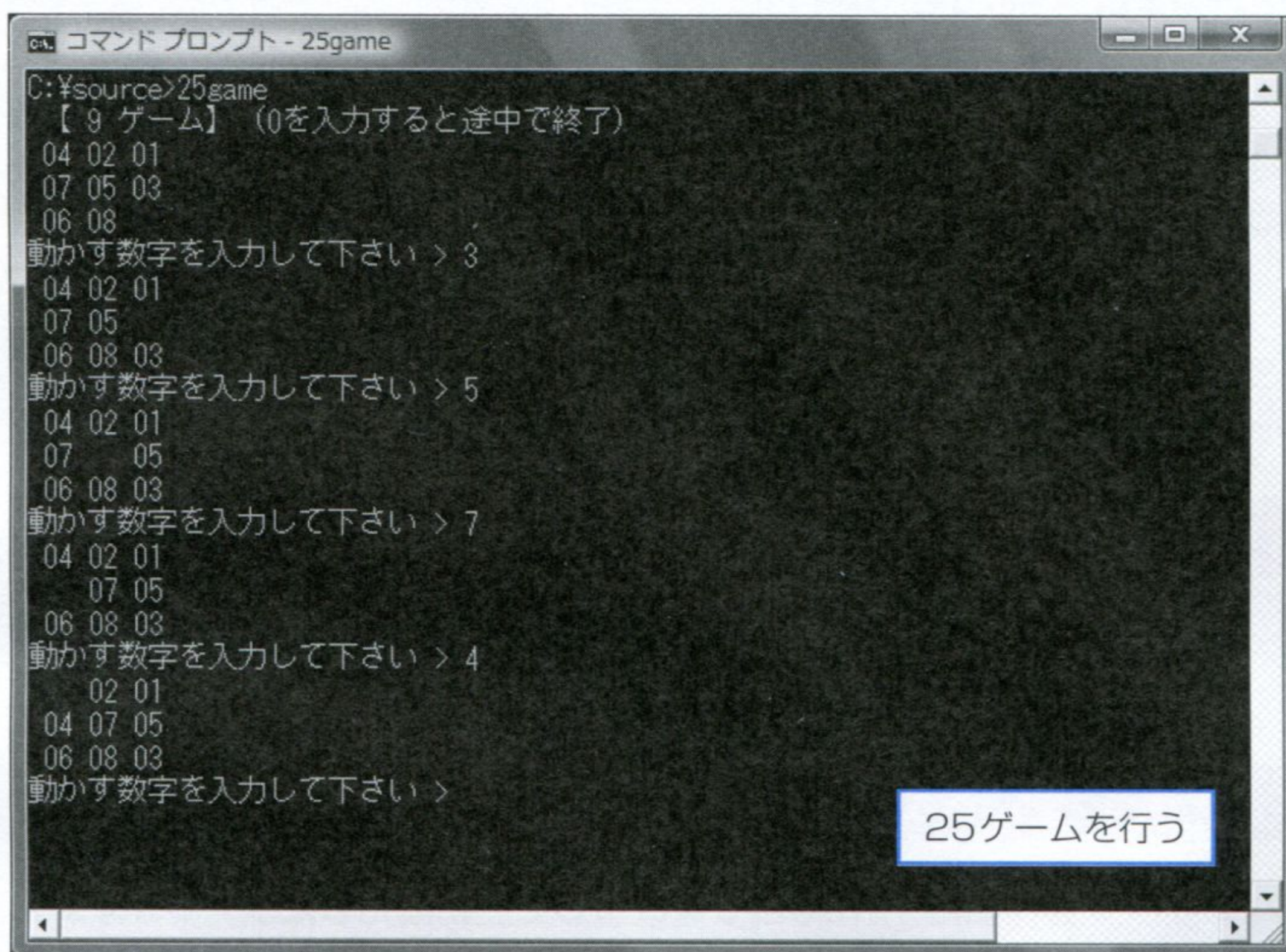
変数

```
tmp_board
{5, 7, 4, 6, 3, 1, 2, 8}
```

解答は巻末に

第9日 3時限目

25ゲームを完成させよう

【25ゲームを作ろう③】

2時限目では、25ゲームの基礎であるゲームのボードを初期化する部分のみを作りました。この時限は、残りの部分を一気に作ってゲームを完成させましょう。

今回作成する例題

```
コマンド プロンプト - 25game
C:¥source>25game
 【 9 ゲーム】 (0を入力すると途中で終了)
 04 02 01
 07 05 03
 06 08
動かす数字を入力して下さい > 3
 04 02 01
 07 05
 06 08 03
動かす数字を入力して下さい > 5
 04 02 01
 07    05
 06 08 03
動かす数字を入力して下さい > 7
 04 02 01
    07 05
 06 08 03
動かす数字を入力して下さい > 4
    02 01
 04 07 05
 06 08 03
動かす数字を入力して下さい >
```

25ゲームを行う

サンプルファイルはこちら 10days_c ▶ day10-03 ▶ 25game.c

●このレッスンのねらい

25ゲームは、これまで作成してきたプログラムよりも複雑なプログラムになります。それなのに、「1時限で残りの部分を作るのは、無理があるのでは？」と思う人もいるでしょう。ですが、先ほど作ったゲームのボードに数値を入れる部分を応用すれば、残りの「数値を動かす」「ボードを表示する」「ボードが完成したかどうか確認する」作業は簡単にできます。

【注意】本レッスンの内容は、本日の2時限目の練習問題の内容を踏まえています。2時限目の練習問題をまだ解いていない人は、ぜひ問題を解いてから本レッスンを学習してください。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define MTX 3
#define CMTX(x) (x)*(x)

// ボード
int board[MTX][MTX];

// 現在空白になっている座標
int b_i;
int b_j;

void initBoard(void);
int moveDigit(int d);
int writeBoard(void);

int main() {
    int d;        // 動かす数値の入力値
    int r = 0;    // リタイアフラグ
    int check;    // 関数 writeBoard からの戻り値
    int count = 0;   // 何回移動で完成できたかカウントする変数

    srand(time(NULL));
    printf("【 %d ゲーム】（0 を入力すると途中で終了）¥n", CMTX(MTX));

    // ボードを初期化する
    initBoard();

    while(1){
        // 移動する数値を入力する
        do {
            d = -1;
            printf("動かす数字を入力して下さい > ");
            scanf("%d", &d);
            while (getchar() != '¥n') {  }
            if(d == 0) { r = 1; break; }
```

```
        } while( d < 1 || d > (CMTX(MTX)-1));

        if(r == 1) { break; }    // リタイアする場合

        if(moveDigit(d) == 0) { continue; } // 数値を移動する

        // ボードが揃ったかどうか調べる
        check = writeBoard();
        count++;
        if(check) { break; }
    }

    // ゲーム結果の表示
    if(check == 1) {
        printf("¥n☆☆☆成功!!☆☆☆¥n");
        printf("移動した回数は %d 回でした¥n",count);
    } else { printf("¥n========終了========¥n"); }

    return 0;
}
// ボードの初期化を行う関数
void initBoard(void) {
    int i, j;
    int m, nokori;
    int r; // ランダムな値
    int digit[CMTX(MTX)-1]; // 残り表示数値を記録する
    int count, k = 0;
    int tmp_board[CMTX(MTX)]; // 表示数値を格納する

    do {
        nokori = CMTX(MTX)-1;
        for(i = 0; i < nokori; i++) { digit[i] = i+1; }
        k = 0;
        for(i = 0; i < CMTX(MTX)-1; i++, nokori--) {
            if(nokori > 1) { r = rand()%nokori; }
            else { r = 0; }      //nokoriの数が1のときはdigit[0]
            tmp_board[i] = digit[r];// ボードに代入する
            for(m = r; m < nokori-1; m++) { digit[m] = digit[m+1]; }
            if(tmp_board[i] != (i+1)) { k = 1; }
        }
    } while(k == 0); // 揃っている状態なら揃えなおし
    tmp_board[CMTX(MTX)-1] = 0;

    count = 0;
```

```
    for(j = 1; j <= MTX; j++) { //縦方向の繰り返し
        for(i = 1; i <= MTX; i++) { //横方向の繰り返し
            board[j-1][i-1] = tmp_board[count];
            if(board[j-1][i-1] != 0) {
                printf(" %02d", tmp_board[count++]);
            }
        }
        printf("¥n");
    }

    b_i = MTX;
    b_j = MTX;
}


/*数値を動かす関数
引数 d：移動する数値
戻り値    0の時は不正な値（移動していない）、1の時は正常移動済
*/
int moveDigit(int d) {
    int i,j;
    int p_i; //動かす数値の入っているマスのx座標
    int p_j; //動かす数値の入っているマスのy座標

    //動かす数値の入っているマスの座標を調べる
    p_i = 0; p_j = 0;
    for(j = 1; j <= MTX; j++) {
        for(i = 1; i <= MTX; i++) {
            if(board[j-1][i-1] == d) {
                p_i = i; p_j = j;
            }
        }
    }

    if(p_i == 0) { return 0; } //入力値が不正な場合

    //数値を移動する
    if(((p_j-1 == b_j) && (p_i == b_i)) ||
        ((p_j == b_j) && (p_i == b_i-1)) ||
        ((p_j+1 == b_j) && (p_i == b_i)) ||
            ((p_j == b_j) && (p_i == b_i+1))) {
        board[b_j-1][b_i-1] = d;
        board[p_j-1][p_i-1] = 0;
    } else { return 0; }
```

```
	// 空白座標の位置を設定しなおす
	b_i = p_i;
	b_j = p_j;
	return 1;
}

/* ボードを表示して完成したかどうかチェックする関数
戻り値	0の時はまだボードは完成していない、1の時は完成
*/
int writeBoard(void) {
	int i, j;
	int d; // 二次元配列の値を取り出して格納する変数
	int check = 0; // チェックのための続き番号
	int rtn = 1; // 完成したかしないかチェックする戻り値

	for(j = 1; j <= MTX; j++) {
		for(i = 1; i <= MTX; i++) {
			d = board[j-1][i-1];
			if(rtn == 1) {
				check++;
				// 数値が一致しない→まだ揃っていない
				if((check < CMTX(MTX)) && (d != check)) { rtn = 0; }
			}
			if(d == 0) { printf("   "); } // 空白の場所の出力表示
			else { printf(" %02d", d); }
		}
		printf("¥n");
	}
	return rtn;
}
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたら、「25game.c」という名前[*1]で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、25game.cをコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o 25game 25game.c
```

4 プログラムを実行する

```
C:¥source>25game.exe
```

```
【 9 ゲーム】(0を入力すると途中で終了)
 03 06 07
 04 08 02
 05 01
動かす数字を入力して下さい > 1
 03 06 07
 04 08 02
 05    01
動かす数字を入力して下さい > 8
 03 06 07
 04    02
 05 08 01
動かす数字を入力して下さい >

            (略)
動かす数字を入力して下さい >
01 02 03
04 05 06
07 08

☆☆☆成功！！☆☆☆
移動した回数は 14 回でした
```

繰り返し数値の移動と、最後にマスがすべて整列しおわったかどうか判定できれば成功！

解説

1 動かす数値を入力する部分を考える

このゲームは、数値を動かさなければ進みません。動かす数値を入力する部分から考えてみましょう。入力部分はmain関数の中に書きます。

数値の入力はこれまでに何回も出てきましたが、数値以外のものが入力されたときの対策を忘れずに行います。また、入力する数値は1～8までなので、それ以外の数値が入力されたときは、何度も入力し直します。しかし、数値以外の文字や文字列が入力されると、scanf関数の入力値を格納する変数には何も入らないので、前に入力した値が入っている状態です。このため、scanf関数を行う前に、入力値を格納する変数に-1を、毎回設定しておきましょう。

```
int d; // 入力値
int r = 0; // リタイアフラグ
while(1){
    do {
        d = -1;
        変数 d に標準入力値を代入 ;
        if(d == 0) { r = 1; break; } //0 が入力されたら終了する

    } while(d < 1 || d > (CMTX(MTX)-1));
    if(r == 1) { break; }

    数値を移動する処理 ;
    ボードが揃ったかチェックする処理 ;
}
```

0が入力されたらリタイアとみなして、繰り返し処理を抜けます。しかし、break文では一番最近の繰り返ししか抜けることはできません。よって、リタイアフラグの変数rを用意しておき、0が入力されたら、rを1に設定します。入力の繰り返しブロックを抜けたあとで再度リタイアフラグを確認し、もし1なら次の繰り返しブロックであるwhile(1)を抜けて、ゲーム全体をリタイアすることになります。

2 数値を動かす自作関数moveDigit

main関数の中で動かす数値が入力されたら、次にその数値を実際に動かす関数を呼び出します。

では、数値を動かすための自作関数moveDigitについて考えてみましょう。

(1) 数値の動かし方

先ほど入力した数値を実際に動かしてみましょう。ボードが下記の状態になっていたとします。

```
【 9 ゲーム】
06 03 07
05 01 04
02 08
```

この場合、動かすことができるのは、空白のマスに隣接する8か4です。よって、8が入力されたら、

```
06 03 07
05 01 04
02    08
```

という状態にする必要があります。これは、今まで空白だった場所board[2][2]に8を設定し、今まで8があった場所board[2][1]に、空白を表す0を設定します。その他に変更はありません。このとき、動かせない場所の数値が入力されたとしましょう。例えば6や3を入力しても、それは現在空白のマスと隣接していないので、動かせません。よって、この場合はboardの変更は行いません。

(2) 関数の引数と戻り値

自作関数moveDigitは入力された数値を動かす関数なので、引数として動かす数値を渡しましょう。この関数は、渡された入力値が動かせない数値だったり不正な数値だったりしたら、0を返すようにします。逆に、正常に数値の移動が行われれば、1を返します。つまり、int型の関数になります。

```
int moveDigit(int d) {
     関数定義 ;
     return 1; // 正常に数値の移動が行われた場合
}
```

main関数の中での呼び出しは、次のようになります。

```
if(moveDigit(d) == 0) { continue; }
```

0のときは数値を動かしてないので、すぐ次の入力を促します。1のときは現在のボードを表示し、同時に今の移動でボードが完成したかどうかチェックする処理を、続けて行います。

(3) 自作関数moveDigitを作る

先ほど簡単に数値の動かし方を説明しましたが、実は、この処理は説明するのは簡単ですが、プログラムで書くと少々複雑になります。しっかり理解してください。

移動する数値と空白の場所を入れ替えればよいだけですが、移動したい数値を入力しただけでは、その数値が入っているマスの場所はわかりません。よって、2次元配列boardに入っている値をすべて見て、その数値が入っているマスの場所を探しあてます。その場所のx座標、y座標を[*2]、変数p_i、p_jにそれぞれ格納します。

```
int i,j;
int p_i; // 動かす数値の入っているマスのx座標
int p_j; // 動かす数値の入っているマスのy座標

p_i = 0; p_j = 0;
for(j = 1; j <= MTX; j++) {
     for(i = 1; i <= MTX; i++) {
          if(board[j-1][i-1] == d) { // 動かす数値と一致した場合 *3
               p_i = i; p_j = j;
```

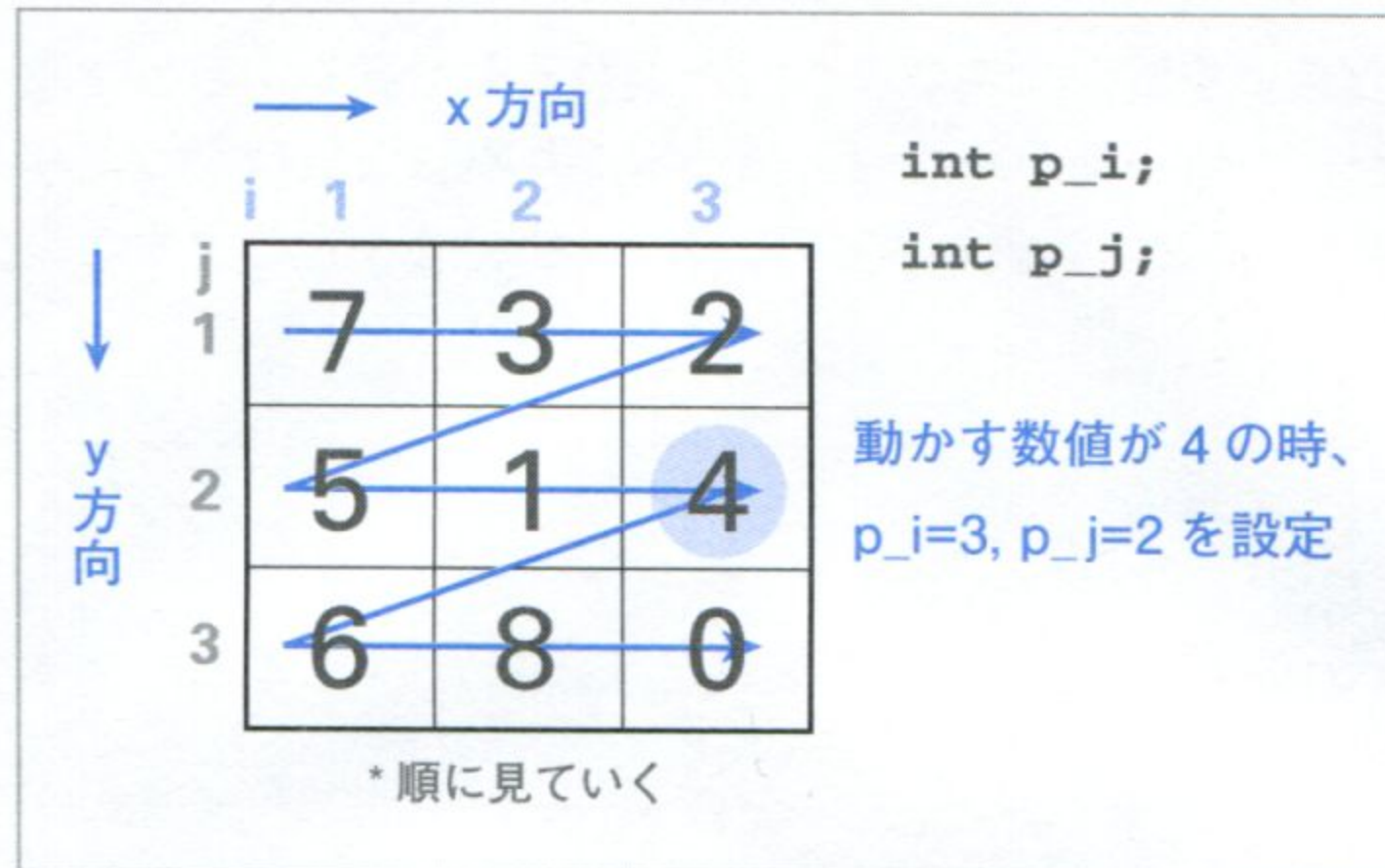
ヒント

*2：x座標、y座標の位置は、1から数えます。配列の添え字（0から数える）ではないので注意してください。

ヒント

*3：動かす数値と一致した場合は、すぐに繰り返しを終了したいところですが、for文を2つ抜けなければなりません。このゲームはマスの数がそれほど多くなることはないので、そのまま最後まで繰り返し処理を行うようにします。

```
            }
        }
    }

    if(p_i == 0) { return 0; }
```

●自作関数moveDigitでの処理イメージ

2次元配列boardの中身をすべて見おわってもp_iが0だった場合[*4]は、boardの中にその数値が見つからなかったということになります。これは不正な値が入力されたことを意味しているので、戻り値0で関数を終了[*5]します。

次に、空白の場所を考えます。動かす数値の場所は調べないとわかりませんが、空白の場所は、x座標もy座標も、最初はMTXの位置ときまっています。数値の移動処理を行ったら、今までその数値があった場所が、今度は空白になります。空白の場所のx座標、y座標をそれぞれ、

```
int b_i; // 空白（0）の入っているマスのx座標
int b_j; // 空白（0）の入っているマスのy座標
```

と定義しておきます。これはプログラムを実行している間ずっと持っておくべき変数なので、グローバル変数にします。空白座標の初期化は、自作関数initBoardの最後に行います。2時限目の練習問題[*6]で作った自作関数initBoardの最後に、次の2行を追加します。

```
void initBoard(void) {
      （略）
      b_i = MTX;
      b_j = MTX;
}
```

自作関数moveDigitの続きに戻ります。動かす数値の場所と空白の場所がわかれば、あとは入れ替えるだけです。ですが、入れ替えるのは、移動することが可能な場合だけです。

ヒント
*4：実際はmain関数の中で入力チェックを行っているので、p_iが0になることはありません。

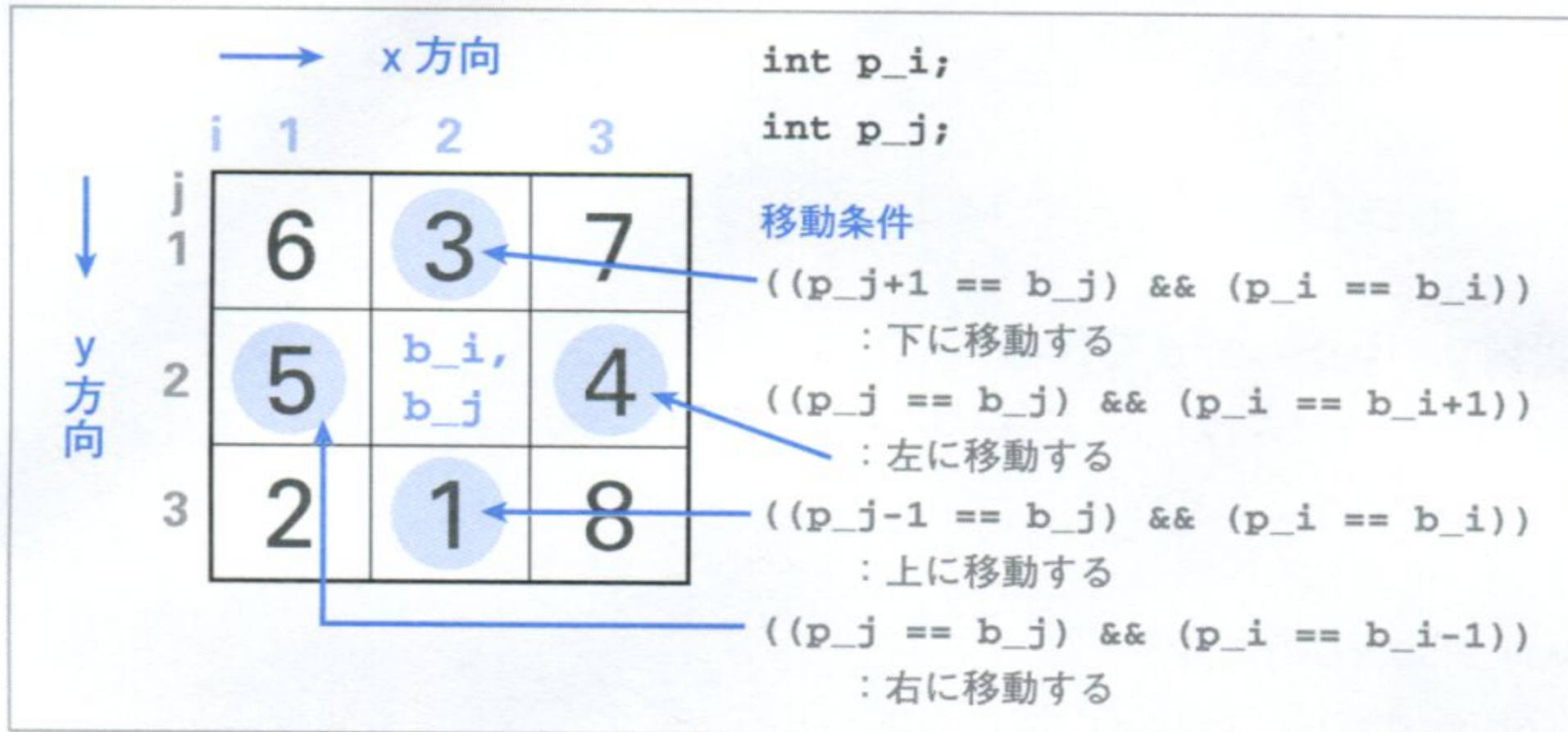
ヒント
*5：関数の途中でreturnを使うと、その場で関数を終了します。

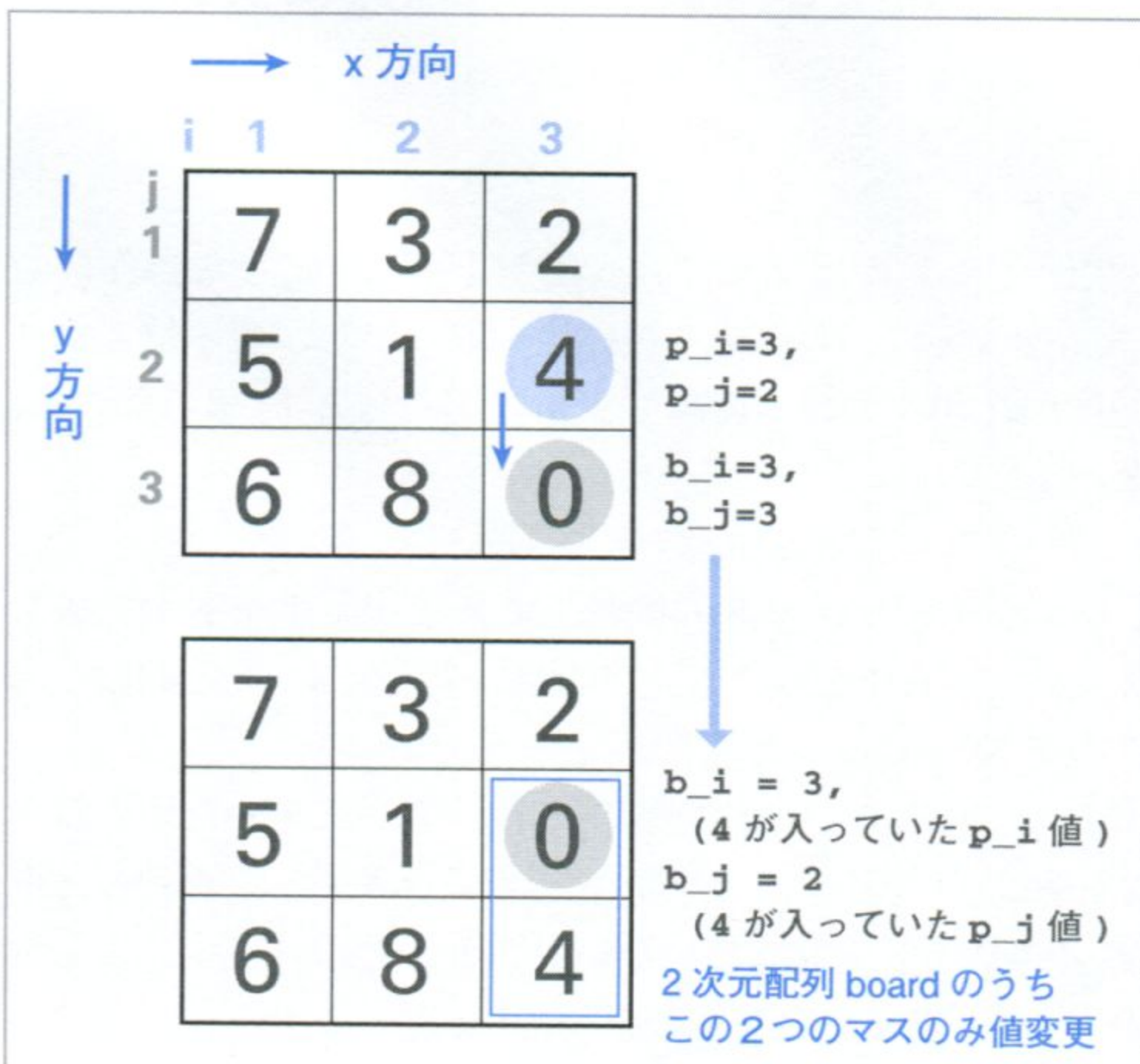
ヒント
*6：第9日2時限目の練習問題をまだ解いていない人は、P375に戻って、ぜひ問題を解いてみましょう。

例えば、マスが次の状態になっていたとします。

```
06 03 07
05    04
02 01 08
```

この状態で動かす数値として6を入力しても、実際に移動することはできません。よって、次の図にあるいずれかの条件にあわなければ、動かせない値を入力したとして、自作関数moveDigitの結果は0になります。

●入力した数値は動かせる？

それ以外なら、空白の座標「x, y」の変数「b_i, b_j」を、新しく空白になった座標位置（直前に動かした数値のあった場所）に設定し、2次元配列boardのうち、この2つのマスのみ値を変更し、1を返して終了します。

●入力した数値を動かす

```
//moveDigit 関数続き
if(((p_j-1 == b_j) && (p_i == b_i)) ||
    ((p_j == b_j) && (p_i == b_i-1)) ||
    ((p_j+1 == b_j) && (p_i == b_i)) ||
        ((p_j == b_j) && (p_i == b_i+1))) {
    board[b_j-1][b_i-1] = d;
    board[p_j-1][p_i-1] = 0;
} else { return 0; } // 動かせない数値の場合

b_i = p_i;
b_j = p_j;
return 1;
```

これで、指定した数値を動かすことができました。

3 自作関数writeBoardを作る

数値の移動がおわったら、現在ボードがどうなっているかを表示します。この処理は自作関数writeBoardとして作成しましょう。

(1) 自作関数writeBoardの概要

自作関数writeBoardでは、移動後の現在のボードの状態を表示すると同時に、数値が1～(MTX*MTX-1)まで順に揃ったかどうか(ボードが完成したかしないか)を判定して結果を返します。

(2) 自作関数writeBoardの引数と戻り値

チェックするボードの2次元配列はグローバル変数として定義されているので、引数は必要ありません。戻り値は、ボードが完成したら1を返します。完成していない場合は0を返します。

(3) 自作関数writeBoardのポイント

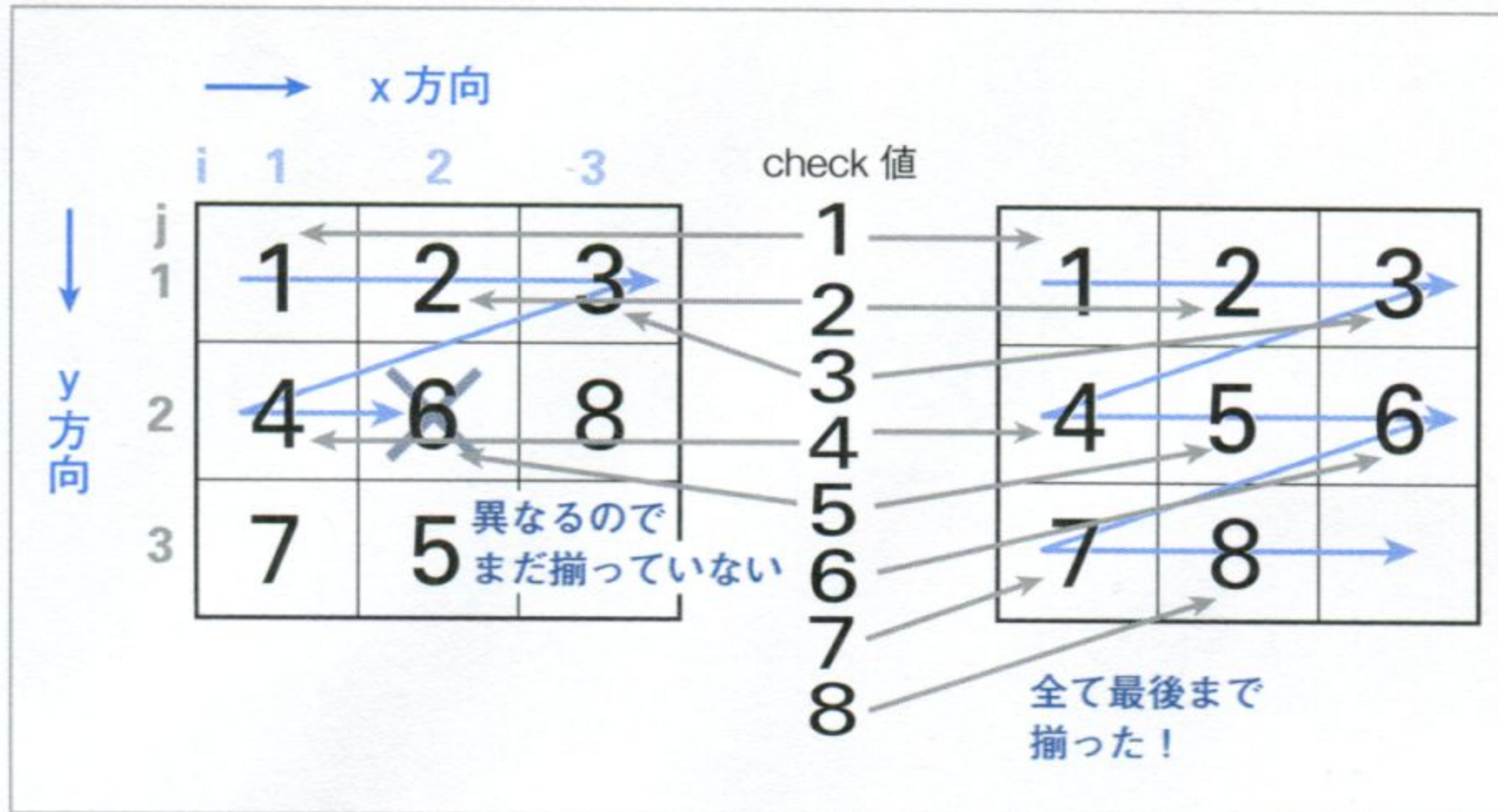
第一の機能である「ボードを表示する」のは簡単です。2時限目で作成した自作関数initBoardを参考に、2次元配列boardをx=1、y=1から順に値を取り出し、表示位置に注意して出力します。出力する桁数は、前に0をつけて埋めて、2桁表示とします。0の部分は、半角スペース2つ分で表示します。

次に、「ボードが完成したかしないかチェックする」機能を考えます。ボードが完成した状態とは、boardに入っている値を端からチェックしたときに、1、2、3……と順に並んでいて、MTX*MTX-1まで空きがなく続いている状態です。

つまり、ボードの表示をするついでに1マス分の値を取り出し、チェックのための続き番号[*7]を格納する変数check(初期値は0)をインクリメントしていきます。この番号と取り出した1マス分の値が一致しない時点で、「完成していない」ことがわかります。もしも最後まですべて一致すれば、「ボードが完成した」ことになり、1を返します。

ヒント

*7:続き番号は、1～(MTX*MTX-1)の間だけチェックします。3×3のときは1～8までです。MTX*MTXも調べてしまうと、もしボードが揃っていた場合9の位置には0が入っているので、一致しなくなります。

●ボードは完成した？

4 25ゲームを完成させる

今まで作った関数*8を組みあわせれば、25ゲームの完成です。ただし、あと少しだけ追加する機能があります。

まず、ボードが完成した場合は「何回数値を動かして完成させたか」と、移動した回数も表示します。その回数は、自作関数writeBoardが呼び出されるたびに加算して、保存しておきます。

```
count++;
```

最後に、main関数の中で結果を表示する部分を作ります。結果は「ボードが完成したか否か」で決定するので、自作関数writeBoardを呼び出した結果が1になった時点で繰り返し処理を強制的に抜け、「成功！」と表示します。

それ以外は途中でリタイアした場合なので、単に「終了」とだけ表示しましょう。

ヒント

*8：自作関数initBoardは、2時限目の練習問題の解答内容を利用しています。

まとめ

2×2だと解答不能な場合があるので、まずは3×3から順に試してみましょう。

第**10**日

○×ゲームを作ろう

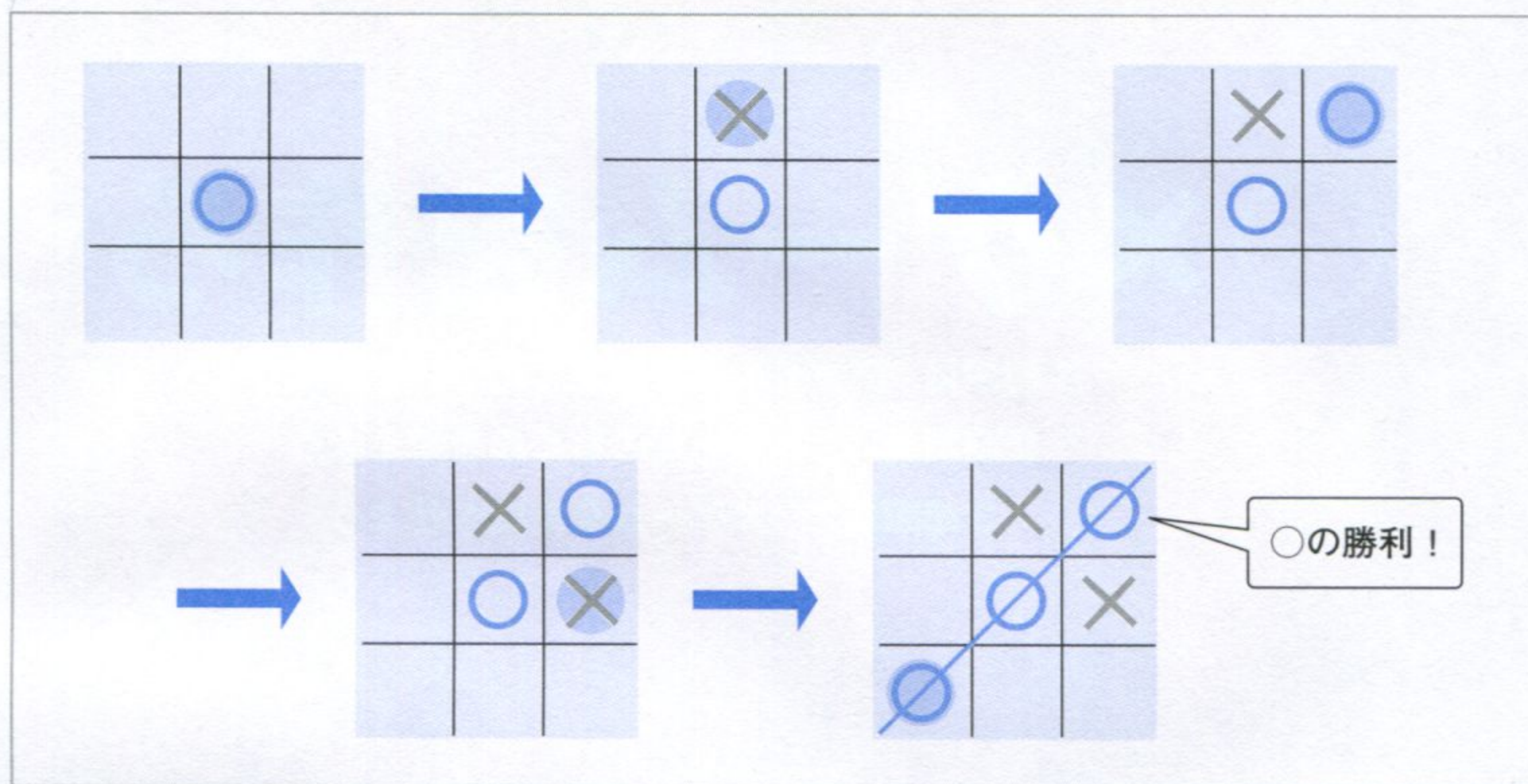
1時限目　ファイルの組み立てについて学ぼう
2時限目　○×ゲームのしくみを考えよう
3時限目　○×ゲームを完成させよう

最終日の本日は○×ゲームを作成します。
関数を多く使うためプログラムが少々長くなるので、プログラムファイルを機能ごとに分割します。
今まで何気なく使ってきたコンパイラですが、コンパイラが実際にどのような作業を行って実行ファイルを作り出しているかを学習し、それを踏まえてプログラムファイルを分割してコンパイルを行う方法学びます。

今日作るプログラムについて

○×ゲームプログラム

○×ゲームは2人で行う、対戦型ゲームです。3×3のマスを作り、空いたマスに両者が○と×を交代で描きます。縦·横·ナナメのうち、1列でも同じマークが揃った方の勝ちです。

○の勝利！

最初にジャンケンプログラムで先攻後攻を決め、プレイヤー（自分）の番になったら、どの位置にマークしたいか、場所を入力します。先攻が○、後攻が×になります。マーク位置は、縦と横の位置を続けて指定します。

```
     [1][2][3]
[a]
[b]      ○
[c]
```

この位置だったら「b2」と指定します。縦方向を示すa、b、cに続けて、横方向を示す数値1、2、3を組みあわせて指定します。すでにマークされている位置を指定することはできません。
対するコンピュータは、プレイヤーがあとひとつでマークが揃う状態にある場合のみ、それを阻止するようにマークします。それ以外のときは、ランダムな位置にマークします。

交互にマークを行い、どれか1列揃うか、またはすべてのマスが埋まってしまったら、ゲームは終了です。最後に勝敗を表示してゲームは終了します。プレイヤーの勝ち・負け・あいこの3種類の結果を表示します。

○×ゲームの実際の動作

1 ○×ゲームを実行すると、ゲームのタイトルが表示される。続いて先攻後攻を決めるジャンケンの手を入力して、[Enter] キーを押す

```
【○×ゲーム】
順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
```

入力する

2 ジャンケンで先攻後攻が決定する。あいこの場合はもう一度ジャンケンを行う

```
【○×ゲーム】
順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ!・・・プレイヤーが先攻(○)です
   [1][2][3]
[a]
[b]
[c]
場所を指定して下さい(例:a2) >
```

先攻の場合は、マークする場所を促すプロンプトが表示される
後攻の場合は、コンピュータがひとつマークを行い、プレイヤーがマークする場所を促すプロンプトが表示される

```
コンピュータはパー!・・・プレイヤーが後攻(x)です
コンピュータは・・・c3
   [1][2][3]
[a]
[b]
[c]          ○
場所を指定して下さい(例:a2) >
```

3 マークする場所を入力して、[Enter] キーを押す。マークする場所がすでに埋まっている場合は、再度入力を行う

```
場所を指定して下さい（例：a2）>b2   ←入力する
    [1][2][3]
[a]
[b]     ○
[c]
```

4 続いてコンピュータがマークする

```
コンピュータは・・・c1
    [1][2][3]
[a]
[b]     ○
[c] ×
場所を指定して下さい（例：a2）>
```

5 上記③～④を、どちらかが1列揃うか、すべてのマスが埋まるまで繰り返す

6 勝敗が決定する

```
場所を指定して下さい（例：a2）> c2
    [1][2][3]
[a] ○  ○  ×
[b] ×  ○  ○
[c] ×  ○  ×

=================== ゲーム終了 =====================
プレイヤーの勝ち！
```

```
コンピュータは・・・c3
    [1][2][3]
[a] ×  ×  ○
[b]    ×  ○
[c]    ○  ○

=================== ゲーム終了 =====================
プレイヤーの負け!
```

```
コンピュータは・・・b1
    [1][2][3]
[a] ×  ○  ×
[b] ○  ×  ○
[c] ○  ×  ○

==================== ゲーム終了 =====================
あいこです
```

第10日 1時限目 【○×ゲームを作ろう❶】ファイルの組み立てについて学ぼう

この時限は、説明を読みながら小さなプログラムを作成し、それを複数のファイルに分割します。

今回作成する例題

10-1_sample1.c - TeraPad

```
1  #include <stdio.h>
2
3  void func0(void);
4
5  int main() {
6      printf("main¥n");
7      func0();
8      return 0;
9  }
10
11 void func0(void) {
12     printf("func0¥n");
13 }
14 [EOF]
```

ひとつのファイルを…

10-1_sample2.c - TeraPad

```
1  #include <stdio.h>
2
3  void func0(void);
4
5  int main() {
6      printf("main¥n");
7      func0();
8      return 0;
9  }
10 [EOF]
```

10-1_func0.c - TeraPad

```
1  #include <stdio.h>
2
3  void func0(void) {
4      printf("func0¥n");
5  }
6  [EOF]
```

複数に分割することもできる

サンプルファイルはこちら 10days_c ▶ day10-01

●このレッスンのねらい

今まで、ひとつのプログラムはひとつのファイルに作成していましたが、ひとつのプログラムを複数のファイルに分割することができます。
プログラムを複数のファイルに分けてコンパイルするには、コンパイラのしくみをよく理解しておく必要があります。この時限では、今まで何気なく使っていたコンパイラのしくみを詳細に説明します。ファイル分割コンパイルを理解して、次の時限から作成する○×ゲームプログラムを、最初から複数のファイルで作成できるようにしましょう。

解説

1 コンパイラのしくみ①：ヘッダーファイルとは

printf関数を使うときは、

```
#include <stdio.h>
```

と書きましょう……と、本書の最初の頃に学習しました。includeは、英語で「～を含む、～を入れる」という意味の単語です。includeのあとに続いて書いたstdio.hはファイル名です。

つまり、この文は「ファイルstdio.hの中身を入れ込む」働きをします。入れ込む場所は、この1文を書いた場所です。

例えば、プログラムの最初に次のように書いてあったら

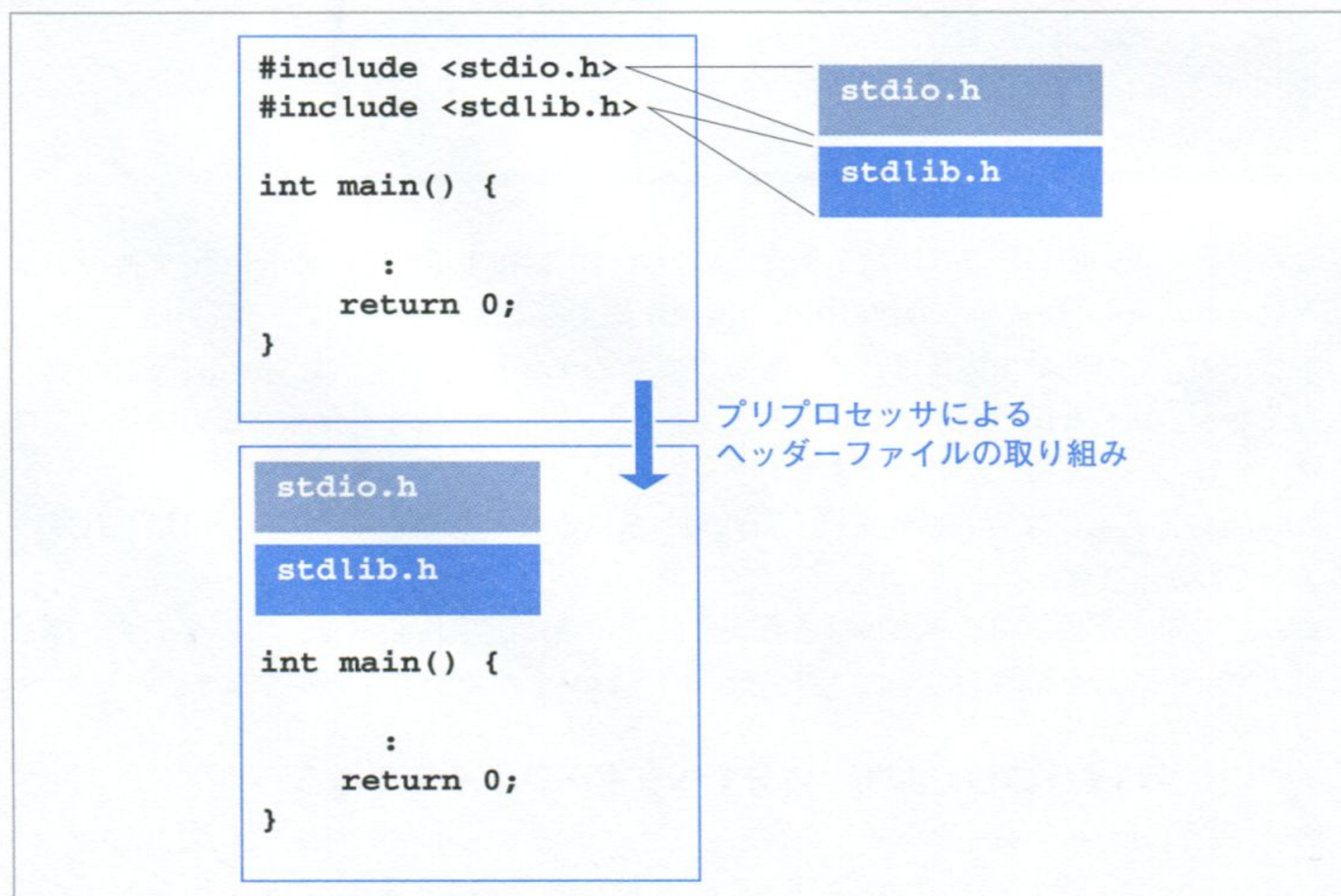
```
#include <stdio.h>
#include <stdlib.h>
```

●インクルード文の役割

```
#include <stdio.h>
#include <stdlib.h>

int main() {

      :
    return 0;
}
```

stdio.h
stdlib.h

プリプロセッサによる
ヘッダーファイルの取り組み

```
stdio.h

stdlib.h

int main() {

      :
    return 0;
}
```

この図のような状態になります。

では、肝心のstdio.hファイルはどこにあるのでしょうか？ 探してみると、以下のディレクトリにあります。

```
C:¥mingw-jp¥include
```

この拡張子「.h」のファイルを、ヘッダーファイルといいます。探し出したstdio.hを開いて見てみましょう。中には関数のプロトタイプ宣言や定数、構造体が定義してあります。

●stdio.hをテキストエディタで開いてみた

```
stdio.h - TeraPad
ファイル(F) 編集(E) 検索(S) 表示(V) ウィンドウ(W) ツール(T) ヘルプ(H)

24  *
25  * $Revision: 1.15 $
26  * $Author: earnie $
27  * $Date: 2003/10/10 15:02:04 $
28  *
29  */
30
31 #ifndef _STDIO_H_
32 #define _STDIO_H_
33
34 /* All the headers include this file. */
35 #include <_mingw.h>
36
37 #ifndef RC_INVOKED
38 #define __need_size_t
39 #define __need_NULL
40 #define __need_wchar_t
41 #define __need_wint_t
42 #include <stddef.h>
43 #define __need___va_list
44 #include <stdarg.h>
45 #endif  /* Not RC_INVOKED */
46
47
48 /* Flags for the iobuf structure  */
49 #define _IOREAD 1 /* currently reading */
50 #define _IOWRT  2 /* currently writing */
51 #define _IORW   0x0080 /* opened as "r+w" */
52
53
54 /*
55  * The three standard file pointers provided by the run time library.
56  * NOTE: These will go to the bit-bucket silently in GUI applications!

1行: 1桁 C/C++ [76] SJIS LF 挿入 書
```

このファイルの中にprintf関数を使うための宣言が存在するので、#includeを使用することにより、printf関数やその他の出入力関数が使えるようになります。

ではそのprintf関数の実体はどこにあるのでしょうか？ C言語では、printf関数の他にもたくさんの便利な関数が用意されています。stdio.hの中を見てもたくさんの関数のプロトタイプ宣言があることがわかります。

stdio.hの他にも、文字列操作を行うstring.hを見ると、たくさんの関数が用意されていることがわかります。

これらの膨大な数の関数の実体はあらかじめ機械語に訳されていて、コンパイラが持っています。それらを組み込んで、プログラムの実行ファイルができあがります。

2 コンパイラのしくみ②：自分でヘッダーファイルを作る

自分でヘッダーファイルを作ることもできます。ヘッダーファイルには、通常、プロトタイプ宣言やマクロ、グローバル変数や構造体が定義されます。自分で作るヘッダーファイルにも、プログラムで使用するプロトタイプ宣言やマクロ、グローバル変数や構造体を書きましょう。

ここでは、第9日2時限目の最後に作ったプログラムのヘッダーファイルを作ってみます。プロトタイプ宣言やマクロ、グローバル変数を抜き出してみましょう。

```
#define MTX 3
#define CMTX(x) (x)*(x)

int board[MTX][MTX];

void initBoard(void);
```

この部分をヘッダーファイル化できます。ここでは、main.hという名前にしましょう。ファイル名は拡張子が「.h」になっていれば、何でもかまいません。

では、このヘッダーファイルをプログラムファイルの中で取り込んでみます。

```
#include <main.h>
```

と書きたいところですが、ファイル名を「<>」で括るのは、C言語の標準関数のヘッダーファイルのみです。自作のヘッダーファイルを取り込む場合は、

【自作のヘッダーファイルの取り込み】

```
#incluce "ファイル名"
```

と、ファイル名をダブルクォート「"」で括ります。このファイル（.hファイル）は、読み込む元のプログラムファイルと同じディレクトリに置きます*1。

プログラムファイルでは、次のように自作ヘッダーファイルを読み込みます。

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#include "main.h"

int main() {
    srand(time(NULL));
    printf("【 %d ゲーム】¥n", CMTX(MTX));
    （略）
}
```

ヒント

*1：正確には自作ヘッダーファイル名の記述は、読み込む元のプログラムのあるディレクトリからの相対パスか、絶対パスで書きます。
標準関数のヘッダーファイルは、コンパイラで定義されている場所（インクルードパスと呼ぶ）からの相対パス指定になります。

自作のヘッダーファイルを利用したプログラムに変更して、もう一度コンパイル、実行をやり直してみてください。問題なくコンパイルされ、実行ファイルができあがると思います。

3 コンパイラのしくみ③：実行ファイルができるまで

ここで、コンパイルがどのようなことを行って実行ファイルを作り出しているか、改めて見てみましょう。

まず、私たちが作るのは、C言語で書いたプログラムのソースファイルとヘッダーファイルです（自作のヘッダーファイルはなくてもかまいません）。これを、

```
C:¥source>gcc -o test test.c
```

とコンパイルしています。すると「test.exe」という実行ファイルができます。

では、今度は-cオプションをつけてtest.cをコンパイルしてみます。

```
C:¥source>gcc -c test.c
```

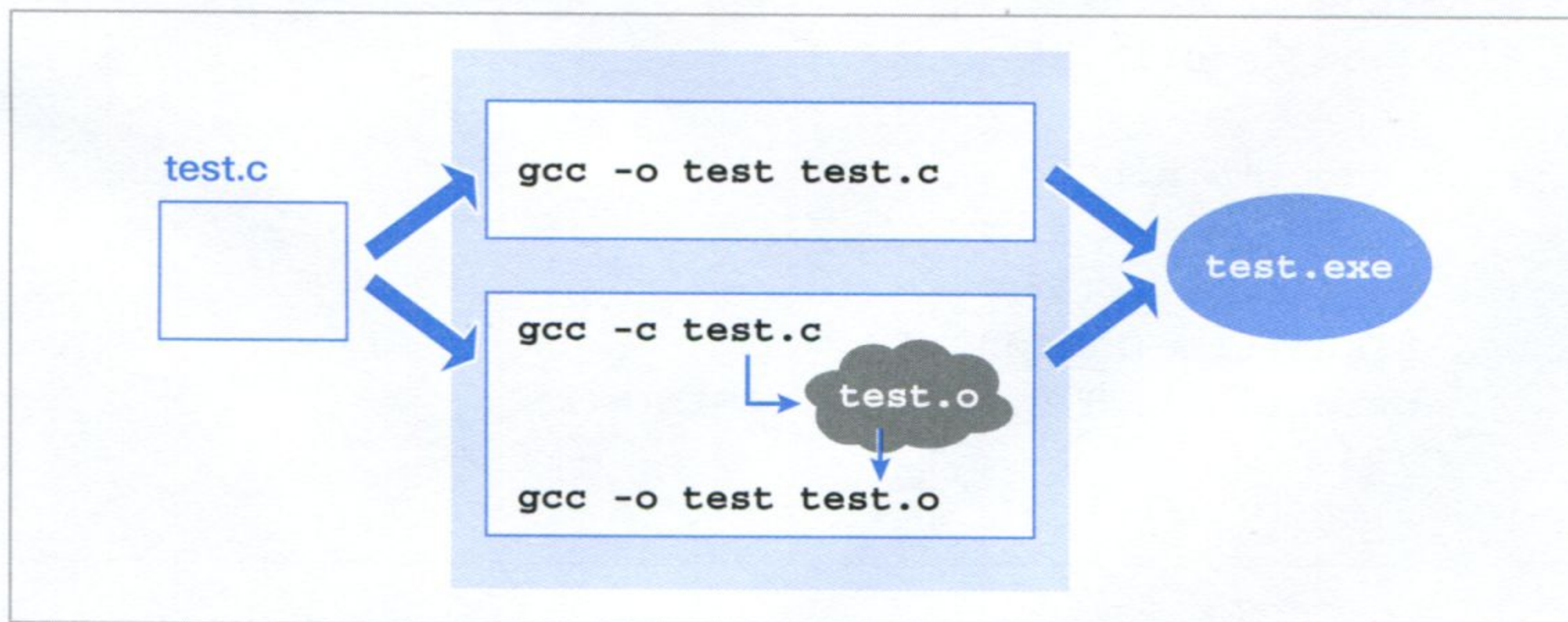
これはtest.cのオブジェクトファイル（test.o）を生成しています。オブジェクトファイルは、C言語の実行ファイルを作る部品のようなファイルです。このtest.oからtest.exeを作るには、次のように実行します。

```
C:¥source>gcc -o test test.o
```

こちらでも、先程と同じtest.exeが作成できたはずです。

この2つの違いはなんでしょうか？ 前者は一度でtest.exeファイルを作成し、後者はオブジェクトファイルという中間ファイルを経て、test.exeを作成しました。実は、実行ファイルができあがるまでは、いくつかの過程があります。前者のように今まで作成してきたプログラムは、その過程を中身の見えないブラックボックスのようにしてコンパイルを行ってきました。後者は、その過程を適当な段階で分けた方法だったのです。

●コンパイルは段階を分けることが可能

test.c
gcc -o test test.c
gcc -c test.c
test.o
gcc -o test test.o
test.exe

では、そのブラックボックスの中をのぞいてみましょう。コンパイラの中で作業をしている機能を紹介します。

●ブラックボックスの中身

・プリプロセッサ
・コンパイラ本体
・リンカ

コンパイラは、この3つの作業過程を経て実行ファイルを作成しています。

C言語では、プリプロセッサはコンパイラ本体に統合されているので、オブジェクトファイルを作成するコンパイル方法では、「コンパイラ本体」までと「リンカ」で2段階に分けて、実行ファイルを作り出したことになります。

この3つの作業過程を、詳細に見ていきましょう。

(1) プリプロセッサ

まず、作成したソースファイルは、プリプロセッサ部分でファイルのインクルードやマクロの置換などを行います。つまり、コンパイル前の準備を行うのがプリプロセッサです。

(2) コンパイラ本体

次に、メインであるコンパイル作業を行います。ソースファイルを、コンピュータが理解できる言語である機械語に変換します。厳密には、この部分のみをコンパイラといいますが、実行ファイルができあがるまでのブラックボックス全体をコンパイラと呼ぶ場合が多いです。

コンパイルを行ったので、各ソースファイルを機械語に変換したファイルができました。これがオブジェクトファイルです。

まだ学習していませんが、複数のC言語のファイルからなるプログラムがあると、そのすべてのファイルは別々にオブジェクトファイルになります。

ソースファイル		オブジェクトファイル
main.c	→	main.o
func01.c	→	func01.o

(3) リンカ

最後に、リンカと呼ばれる部分を実装します。コンパイラで作成されたオブジェクトファイルは、自分で作ったプログラム部分しか機械語に変換されていません。中で使っているprintf関数やrand関数などのC言語の標準関数は、ヘッダーファイルは取り込みましたが、肝心の中身の部分は別のファイルに入っているので、このままでは標準関数部分は実行することができません。

C言語の標準関数の中身部分は、すでに機械語になったファイルとして用意されています。これをライブラリファイルといいます。

リンカとは、「linker(つなげる)」機能を持つ部分です。コンパイラで作成されたオブジェクトファイルと、このライブラリファイルの必要な部分だけをつなげ（リンクするといいます）、拡張子.exeの実行ファイルを作りあげます*2。

ヒント

*2：オブジェクトファイルが複数ある場合も、ひとつの実行ファイルにまとめます。

●ブラックボックス内の作業

プログラムファイル
main.c

```
#include <stdio.h>
#include "main.h"

#define MTX 3

int main() {
     int board[MTX];
   :

}
```

ヘッダーファイル
main.h

```
int func01(void);
```

プログラムファイル
func01.c

```
#include <stdio.h>

int func01(void) {
   :
}
```

Cコンパイラ

①プリプロセッサ

stdio.h

main.h

```
int main() {
     int board[3];
   :

}
```

準備する

stdio.h

```
int func01 (void) {
   :
}
```

②コンパイラ

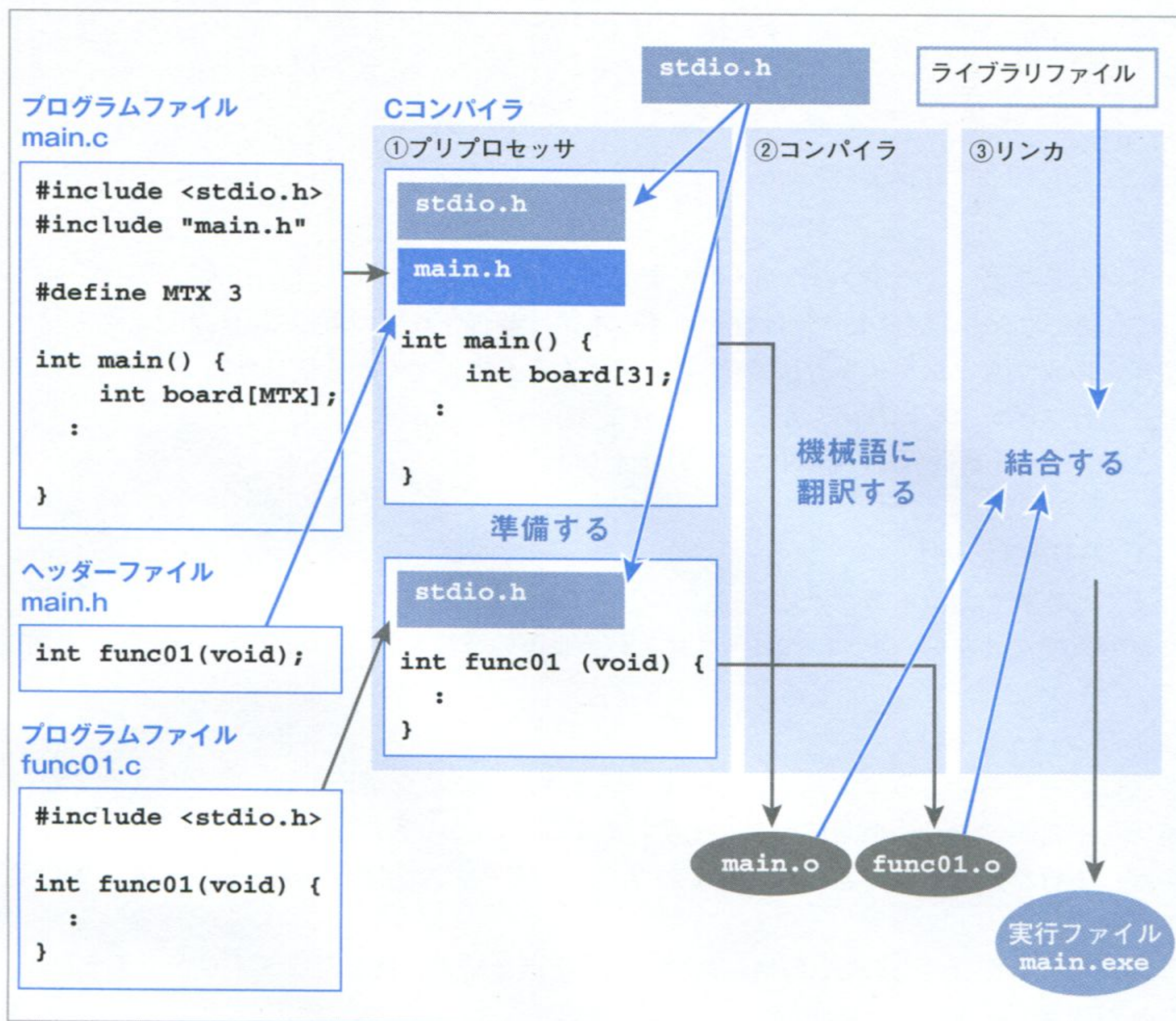
機械語に
翻訳する

③リンカ

ライブラリファイル

結合する

main.o

func01.o

実行ファイル
main.exe

作成された実行ファイルは、C言語の標準関数部分も含め、すべて必要な機能だけを機械語にしてあるファイルです。このため、まったく同じ環境のマシンならば、実行ファイルだけでそのプログラムを動かすことができます。

難点は、他のテキストファイルに比べると容量が大きいということですが、インタプリタ言語と違って実行するための機能（インタプリタ）が必要ないので、実行時の速さと移植性を考えれば、やはりC言語で作った実行ファイルの方が、メリットが大きいといえるでしょう。

4 プログラムファイルの分割①：複数のファイルからなるプログラム

複数のC言語のソースファイルから、ひとつの実行ファイルを作り出すこともできます。まずはmain関数とひとつの自作関数func0からなるプログラムを、ひとつのファイルで作成してみます。

【10-1_sample1.c】

```
#include <stdio.h>

void func0(void);

int main() {
    printf("main¥n");
    func0();
    return 0;
}

void func0(void) {
    printf("func0¥n");
}
```

```
C:¥source>10-1_sample1.exe
main
func0
```

main関数から自作関数func0を呼び出すことができます。このファイルを10-1_sample1.cとします。では、この自作関数func0を別のファイルに書いて、プログラムを分割してみましょう。

(1) ファイルを分割する

まず、ファイルをmainとfunc0に分割します。10-1_sample1.cのコピーファイルを2つ作り、それぞれ10-1_sample2.cとfunc0.cとします。そして、お互い余計な部分を消しましょう。

どちらのファイルでもprintf関数を使っているため、stdio.hを両方にインクルードします。

【10-1_sample2.c】

```
#include <stdio.h>

void func0(void);

int main() {
    printf("main¥n");
    func0();
    return 0;
}
```

【10-1_func0.c】

```
#include <stdio.h>

void func0(void) {
    printf("func0¥n");
}
```

これをコンパイルします。Cのソースファイルは分割されてしまったので、2つともコンパイルする必要があります。作成した2つのファイルを引数に、次のようにしてコンパイルを実行しましょう。

```
C:¥source>gcc -o 10-1_sample2 10-1_func0.c 10-1_sample2.c
```

10-1_sample2.exeができあがったはずです。func0.cの機械語のコードは、リンカ機能により、ちゃんと10-1_sample2.exeの中に含まれています。

できあがった10-1_sample2.exeを実行すると、分割する前の10-1_sample1.exeと同じ結果になります。このようにソースを分割してコンパイルを行うことを、分割コンパイルといいます。

(2) ヘッダーファイルを使う

今度はヘッダーファイルも使ってみましょう。ヘッダーファイルにはグローバル変数や関数のプロトタイプ宣言を書いて、cファイルから読み込んで使います。

10-1_sample2.cをコピーして10-1_sample3.cを作り、その中にある自作関数func0のプロトタイプ宣言を10-1_func0.hファイルに書き、main関数のあるファイルの中でincludeを使って読み込みます。

【10-1_sample3.c】

```
#include <stdio.h>

#include "10-1_func0.h"

int main() {
    printf("main¥n");
    func0();
    return 0;
}
```

【10-1_func0.h】

```
void func0(void);
```

10-1_func0.cの内容はかわりません。
先程と同様に、分割コンパイルを実行してみます。

```
C:¥source>gcc -o 10-1_sample3 10-1_func0.c 10-1_sample3.c
```

できあがった10-1_sample3.exeを実行すると、この方法でも同じ結果になることが確認できると思います。

実際にプログラムを書くときは、今のように最初から存在するプログラムを分割することはあまりありません。プログラムを作るときにあらかじめ、

・この機能は別のファイルに書こう ——→ 10-1_func0.cの作成
・自作関数を使うためにmainのあるファイルではプロトタイプ宣言を書こう
・プロトタイプ宣言はヘッダーファイルにしよう ——→ 10-1_func0.hの作成

とおおまかな構成を考えてプログラムを作ります。
プログラムの作成途中で機能が増えたり減ったりすることもあるので、これはあくまで目安です。「だいたいこんなファイル構成になる……かな？」ぐらいに考えて作れば大丈夫です。

5 プログラムファイルの分割②：分割コンパイルの変数の有効範囲

グローバル変数は、ファイルの中全体で有効になる変数です。しかし、ソースファイルを分割した場合はどうなってしまうでしょうか？

(1) 複数のソースファイルでグローバル変数を使う

main関数のある10-1_sample4.c*3の中で、グローバル変数dを定義します。
そして、10-1_sample4.cの中で定義されているこのグローバル変数dを、自作関数func1でも使ってみます。

ヒント
*3：今回は、自作ヘッダーファイルは使用しません。

【10-1_sample4.c】

```
#include <stdio.h>

void func1(void);

int d = 10;

int main() {
    printf("main d:%d¥n", d);
    func1();
    return 0;
}
```

【10-1_func1.c】

```
#include <stdio.h>

void func1(void) {
    d++;
    printf("func1 d:%d¥n", d);
}
```

ですが、コンパイルを実行するとエラーになります。自作関数func1のファイルには変数dがどこにも定義されないので、エラーになるのです。

```
C:¥source>gcc -o 10-1_sample4 10-1_func1.c 10-1_sample4.c
10-1_func1.c: In function `func1':
10-1_func1.c:4: error: `d' undeclared (first use in this function)
10-1_func1.c:4: error: (Each undeclared identifier is reported
only once
10-1_func1.c:4: error: for each function it appears in.)
```

10-1_func1.cの中でもグローバル変数dを使いたい場合は、次のように定義してみましょう。

```
#include <stdio.h>

extern int d;  //10-1_sample4.cの中で定義されている変数dを使う

void func1(void) {
    (略)
```

externをつけて宣言した変数は、実体が他のファイルにあるという意味になるので、その場所を探して参照します。参照するだけでなく、func1関数の中では「d++」という処理を行ったので、変数dの値は変更されます。

ただし、これはコンパイラの種類にもよるもので、gccではexternがなくても変数dを共有しています。他のコンパイラでは、externをつけないと共有せず、別の変数として扱われます。もし意図的にではなく別ファイルのグローバル変数が同じ名前になったときに値が共有されると、困ったことになります。

-fno-commonオプションをつけてコンパイルするとexternなしではエラーになるので[*4]、ファイルを分割して共有のグローバル変数を使うときは、このオプションをつけるようにしましょう。

実行してみます。

```
C:¥source>gcc -fno-common -o 10-1_sample4 10-1_func1.c 10-1_sample4.c
```

▼

```
main d:10
func0 d: 11
```

10-1_sample4.cの中にある変数dの値が、10-1_func1.cでも利用できました[*5]。

(2) グローバル変数にstaticを使うとどうなるか

今度は、10-1_sample4.cの変数dの宣言の前に、staticをつけたらどうなるか、試してみましょう。

```
static int d = 10
```

しかし、コンパイルするとエラーが出ます。変数の前にstaticとつけると、その変数の有効範囲は宣言したファイルに限られます。この変数を、静的変数と呼びます。よって、他のファイルからexternして使おうとしても、「使えないよ」とコンパイラにいわれてしまうのです。

外部ファイルからの参照を行いたいグローバル変数には、staticをつけないようにしましょう[*6]。

(3) staticの使い方

ローカル変数に使うと、staticは非常に便利なときがあります。

グローバル変数を使わない、簡単な関数呼び出しプログラムを書いてみます。

ヒント

*4：externなしでグローバル変数を両方初期化すると、コンパイルエラーになります。

ヒント

*5：他のファイルに実体がある関数のプロトタイプ宣言も、本来ならexternをつけます。ただし、「extern void func1(void);」のexternは省略可能です。

ヒント

*6：関数宣言にもstaticをつけることができます。そうした場合、その関数の有効範囲は、定義されているファイル内のみです。

【10-1_sample5.c】

```
#include <stdio.h>

void func2(void);

int main() {
    func2();
    func2();
    return 0;
}
```

【10-1_func2.c】

```
#include <stdio.h>

void func2(void) {
    int d = 0;
    d++;
    printf("func2 d: %d¥n", d);
}
```

自作関数func2を2回呼び出すプログラムです。コンパイルして実行してみます。結果は次のようになります。

```
func2 d: 1
func2 d: 1
```

では、自作関数func2の中の変数dを、static変数に変えてみます。

```
static int d = 0;
```

こちらの実行結果は、次のようになります。

```
func2 d: 1
func2 d: 2
```

変数dの値が1回目と2回目で異なります。これは、staticをつけたローカル変数はプログラムの実行時に作られて、プログラムが終了するまで値が保持されるからです。

変数dの初期値は0なので、最初にfunc2が呼び出されたときにインクリメントされて、1になります。変数dはstatic変数なので、次にfunc2が呼び出されたときには新しい変数領域は作られず、最初に作った変数dの値が参照されてインクリメントします。よって、次は2になりました。

こういった方法でのstatic変数は小さいプログラムではあまり使う必要はありませんが、何度も何度も同じ関数を呼び出すグラフィックゲームのプログラムには、よく使われます*7。

ヒント
*7：この使い方は、特に分割コンパイルの場合のみに限った使い方ではありません。

まとめ

この時限では、ヘッダーファイルの使い方や、分割コンパイルの方法を学習してきました。今のところはどれも小さいプログラムなので、分割してもあまり利点がないように思うかもしれません。しかし、今後C言語にどんどん慣れていき、とても重いプログラムを書くこともあるでしょう。長大なソースコードになってしまうと、書くのもあとで見直すのも、一苦労です。

加えて、大きいプログラムになると、数人、規模によっては数十単位の人が手がけることもあります。本書の読者のみなさんのほとんどは、C言語をはじめて10日前後のはずですので、そうした案件を手がけることになるのはかなり先の話になるかもしれませんが、これまでのレッスンで徐々に長いプログラムが書けるようになってきたのは間違いありません。

構造体やプロトタイプ宣言などを一度定義したら、あとは場所を取るだけの宣言文をヘッダーファイルに保存しておけば、main関数についてはファイルを開けばすぐわかる位置に表示されます。

また、ソースファイルをプログラムの機能ごとに分割しておけば、デバッグがとてもしやすくなります。

さらに複数人でプログラムを書く場合、それぞれが作った関数をひとつのファイルに最後に統合して編集し直すよりも、そのままの別のファイルで保存しておいたほうが、各自が他人に迷惑をかけることなく、修正することができます。

と、ここまではプログラムを書く側の利点を述べました。コンピュータの側から見た利点についても述べておきましょう。

分割コンパイルを行うと、変更が加えられていないファイルでは、再度コンパイルしてオブジェクトファイルを作る必要がありません。修正したファイルのみをコンパイルすればよいので、コンピュータへの負荷が減ります。

そして、プログラムファイルを分割し、変数の種類（static、local、externなど）を上手に割り振ると、実行速度が上がります。このあたりは慣れとコツが必要になりますが、大きいプログラムでは、ソースの書き方によってかなり実行速度が違ってくるのです。

今の段階では、まだ「機能ごとに分けて見やすくする」「管理を容易にする」が唯一の利点かもしれません。2時限目からは、この分割コンパイルを使って、プログラムを書いてみましょう。

第10日 2時限目

【○×ゲームを作ろう②】○×ゲームのしくみを考えよう

この時限では、○×ゲームの機能の洗い出しを行い、ゲーム序盤までを作成します。

今回作成する例題

```
C:¥source>marubatu.exe
【○×ゲーム】
順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはグー！・・・もう一回
順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー！・・・プレイヤーが後攻（×）です
コンピュータは・・・
   [1][2][3]
[a]
[b]
[c]

===================ゲーム終了=====================
プレイヤーの負け！

C:¥source>
```

ジャンケンで先攻の方を勝ち・後攻を負けと表示する

サンプルファイルはこちら　10days_c ▶ day10-02 ▶ marubatu.c

●このレッスンのねらい

1時限目でファイルの分割コンパイルについて学びました。ソースファイルは、プログラムが長くなったときに機能ごとに分割すると便利です。
実は、○×ゲームはそれほど長いソースにはならないのですが、分割コンパイルを実践で使ってみるために、ここではファイルを分割して作成しましょう。
最初にゲームの機能を洗い出し、まずは関数名とファイルの分割方法をきめ、main関数の大部分とゲーム序盤で使用する関数のみを作成します。その他の関数は、処理を書かない状態で用意しておきます。

プログラムを作成する

1 テキストエディタで新規文書を作成し、次のコードを入力する

【marubatu.c】

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include "marubatu.h"
#include "marubatu_func.h"

//○×ゲーム
int main() {
    int p_turn; //プレイヤーターンのときは1
    char p_get[3]; //プレイヤーの入力値
    int r;         //マークが揃ったかどうかの判定結果

    srand(time(NULL));
    printf("【○×ゲーム】¥n");
    do { p_turn = janken(); } while(p_turn < 0);
    if(p_turn) { writeBoard(); }

    for(;; p_turn = !p_turn) {
        if(p_turn){ //プレイヤーターン
            do {
                printf("場所を指定して下さい（例：a2）> ");
                scanf("%s", p_get);
            } while(!writeMark(p_get));

        } else {     //コンピュータターン
            writeCompMark();
            printf("コンピュータは・・・¥n");
        }
        writeBoard();
        r = judge();
        if(r != 0) { break;}
    }
   printf("¥n============== ゲーム終了 ===============¥n");
   if(r == -1) { printf("あいこです¥n"); }
   else {
        if (p_turn) { printf("プレイヤーの勝ち！¥n"); }
        else { printf("プレイヤーの負け！¥n"); }
```

```
    }

    return 0;
}
```

【marubatu_janken.c】

```
#include <stdio.h>
#include <stdlib.h>
#include "marubatu.h"

int p_maru = BATU;   // プレイヤーが○の時は MARU、×の時は BATU

/* ジャンケンでゲームの先攻後攻をきめる関数
戻り値    r    1：プレイヤーの先攻
               0：プレイヤーの後攻
              -1：あいこでもう一回勝負
*/
int janken(void) {
    int r; // 戻り値
    int player, computer; // プレイヤーとコンピュータの手
    char *shurui[] = { "グー", "チョキ", "パー" };

    do {
        player = 0;
        printf("順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > ");
        scanf("%d", &player);
        while (getchar() != '¥n') {  }
    } while(player < 1 || player > 3);
    computer = rand()%3 + 1;
    printf("コンピュータは%s!・・・", shurui[computer-1]);

    if(computer == player) {
        printf("もう一回¥n");
        r = -1;
    } else if((player == 1 && computer == 2) ||
            (player == 2 && computer == 3) ||
            (player == 3 && computer == 1)) {
        printf("プレイヤーが先攻(○)です¥n");
        p_maru = MARU;
        r = 1;
    } else {
        printf("プレイヤーが後攻(×)です¥n");
```

```
        r = 0;
    }
    return r;
}
```

【marubatu_board.c】

```
#include <stdio.h>
#include <stdlib.h>
#include "marubatu.h"

static int board[MTX][MTX]; // ボード配列

/* ボードを表示する関数 */
void writeBoard(void) {
    int i, j;
    char *mark[] = { "  ", "○", "×"};

    printf("   ");
    for(i = 1; i <= MTX; i++) {
        printf("[%d]",i);
    }
    printf("¥n");
    for(j = 0; j < MTX; j++) {
        printf("[%c]", j+'a');
        for(i = 0; i < MTX; i++) {
            printf("%3s", mark[board[j][i]]);
        }
        printf("¥n");
    }
}

/* プレイヤーの選択した位置にマークをする関数
引数 str*      プレイヤーが指定したマーク位置文字列
戻り値   r     1：正常な値だったのでマーク完了
               0：不正な値だったのでマークできなかった
*/
int writeMark(char* str) {
    int r = 1;

    return r;
}
```

```
/* コンピュータの選択した位置にマークをする関数 */
void writeCompMark(void) {
}

/* マークが揃ったかどうかをチェックする関数
戻り値   r    1：マークが揃っている
              0：マークはまだ揃っていない
             -1：全てのボードが埋まったがどちらのマークも揃っていない
*/
int judge(void) {
    return 1;
}
```

【marubatu.h】

```
#define MTX 3 /* マス目の数（縦横）最大9まで */

#define MARU 1
#define BATU 2
```

【marubatu_func.h】

```
void writeBoard(void);
int writeMark(char* str);
void writeCompMark(void);
int judge(void);

int janken(void);
```

ヒント

*1：拡張子に注意して保存しましょう。

2 入力できたらそれぞれの名前*1（手順①の【】内参照）で、「C:¥source」ディレクトリ下に保存する

3 コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、ソースファイルを3つともコンパイルする

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o marubatu marubatu_janken.c marubatu_board.c marubatu.c
```

4 プログラムを実行する。ジャンケンで先攻を取ったほうが自動的に勝ち、後攻の場合は自動的に負けになれば成功！

```
C:¥source>marubatu.exe
```

```
【○×ゲーム】
順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはチョキ！・・・プレイヤーが先攻（○）です
   [1][2][3]
[a]
[b]
[c]
場所を指定して下さい（例：a2） > a1
   [1][2][3]
[a]
[b]
[c]

==================== ゲーム終了 =====================
プレイヤーの勝ち！
```

```
【○×ゲーム】
順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー！・・・プレイヤーが後攻（×）です
コンピュータは・・・
   [1][2][3]
[a]
[b]
[c]

==================== ゲーム終了 =====================
プレイヤーの負け！
```

解説

1 ○×ゲームの概要

○×ゲームは3×3のマスを作り、2人で交互に○と×を書いていきます。縦・横・ナナメのどれか1列に、先に同じマークが揃った方の勝ちです。

プレイヤー対コンピュータの対戦ゲームで、先攻が○、後攻が×を交互に書きますが、○を取った方が有利なので、ゲームの最初に先攻後攻を公平にジャンケンできめます。

先攻後攻がきまったら、○×を書いていきます。プレイヤーは場所の指定を入力します。コンピュータの場合はルールに従ってきめられた場所を指定します。

これを繰り返し、どちらかが1列揃えば終了です。揃わなければあいこです。

見た目は簡単ですが、コンピュータ側も多少思考をしなければならないので、今までよりはボリュームのあるプログラムになります。

2 ○×ゲームに必要なプログラムの機能を洗い出す

次に、このゲームを実現するためにどんな機能が必要か、考えてみます。

●○×ゲームに必要な機能

①先攻後攻をきめる機能
・ジャンケンゲームをして、勝ったほうが先攻（○）を取る
②○×を描く機能
・プレイヤーの場合は場所を入力してもらい、その場所で問題がなければマークする
・コンピュータの場合はランダムか、場合によっては思考して、場所をきめてマークする
③現在のボード[*2]の状態を出力する機能
④両者どちらの場合でも、１列揃ったかどうかを判定する機能
・揃うかすべてのマスが埋まったら、ゲームを終了する
⑤勝敗を判定する機能

ヒント
*2：このプログラムでは、○×を書いた３×３のマスをボードと呼びます。

このうち、②〜④までは繰り返し処理です。この繰り返し部分はmain関数の中で行います。プログラムのすべてをmain関数の中で書くと長くなってしまうので、特定の処理部分は関数化します。

(1) 自作関数janken

まず①のジャンケンゲームですが、これは本書の中ですでに何回か出てきました。ここでは関数化しましょう。

(2) 自作関数writeBoard

プレイヤーが先攻の場合は、ボードがどんな状態なのかわからないので、一度ボードを表示します。これは、○×を入力したあとでも同じ機能が必要になるので、関数にします。

(3) 自作関数writeMark

プレイヤーがボードに○か×をマークする関数を作ります。

(4) 自作関数writeCompMark

一方、コンピュータがマークするときの処理はまったく異なるので、(3) とは別に関数を作ります。

(5) 自作関数judge

1列揃ったかどうかを判定します。

考えられる関数は、この5つです。プログラムの作り方は人それぞれですが、今回はこの5つを関数化しましょう。(1) の自作関数janken以外は、すべて○×を記録したボードを参照したり、マークしたりしています。よってソースファイルを次の3つに分けます*3。

● ○×ゲームプログラムのソースファイル分割方法

・メインプログラムファイル（marubatu.c） → main
・ジャンケン関数ファイル（marubatu_janken.c） → janken
・ボード関数ファイル（marubatu_board.c）
　→ writeBoard、writeMark、writeCompMark、judge

ヒント
*3：ファイルの大きさに違いが出てしまいますが、今回は使用する機能で分けるのでこれでかまいません。

3 ○×ゲームの構成について考える

関数について、それぞれ見ていきます。

(1) main関数

main関数で行うことは、さきほど機能を洗い出したときに列挙した5項目のすべてです。つまり、今回作るすべての関数は、main関数から呼び出します。プログラム的に見てみましょう。main関数の処理内容は次のようになります。

●main関数での処理

```
int p_turn; //プレイヤーターンのときは1

ジャンケンを行って先攻後攻をきめる　→　janken()
プレイヤーが先攻の時はボードを表示する　→　writeBoard()

for(;; p_turn = !p_turn) {
    if(p_turn) {
        正しい場所にマークするまで場所を読み込む→writeMark(プレイヤー入力値)
    } else {
```

```
        コンピュータがマークする  →  writeCompMark()
        コンピュータがマークした場所を表示する
    }
    ボードを表示する  →  writeBoard()
    判定を行う  →  judge()
    if(どこか揃ってたら) { break; }
}

勝敗結果を判定して表示する;
```

関数の処理内容は、考え方によりさまざまな書き方があります。

例えば先攻後攻をきめるとき、自作関数jankenをmain関数の中で1回だけ呼び、先攻後攻がきまるまでの繰り返し処理はその自作関数jankenの中で行うか、自作関数jankenの中身は1回勝負のみで勝負の判定はmain関数側で行うか、関数の中でどこまで書くかにより呼び方も異なります。

●関数の中で先攻後攻がきまるまですべて行う場合の呼び出し方

```
p_turn = janken();
    janken()は
        プレイヤーが勝てば1
        コンピュータが勝てば0  を返す
```

●関数の中身は1回ジャンケン勝負がつくまで。あとはmainで制御する方法

```
do { p_turn = janken(); } while(p_turn < 0);
    janken()は
        プレイヤーが勝てば1
        コンピュータが勝てば0
        あいこなら-1            を返す
```

プレイヤーがマークする場所を指定するときも同じです。不正な場所を指定したときに正しい場所を入力するまで繰り返す処理を、mainの中で行うか関数の中で行うか、ということです。

今回作る関数は基本的に処理のみを行い、繰り返しの制御はすべてmain関数で行うようにします。このようにすると、main関数を見ただけでプログラムの全体がよくわかります。

逆に、次のようにすべての処理を各関数の中におさめてしまうこともあります。

```
int main() {
    func1();
    func2();
    func3();
    return 0;
}
```

この例の場合は関数名が適当なので、この例の場合は何をやっているかわからないプログラムになりますが*4、関数名を処理内容に基づいてつければ問題ありません。

(2) 関数を定義する

関数定義ファイルを作成しましょう。

ジャンケンを行う自作関数janken*5は、marubatu_janken.cファイルに作成します。

【marubatu_janken.c】

```
/* ジャンケンでゲームの先攻後攻をきめる関数
戻り値    r    1：プレイヤーの先攻
               0：プレイヤーの後攻
              -1：あいこでもう一回勝負
*/
int janken (void) {
    int r; // 戻り値
    return r;
}
```

他4つのボード関係の関数は、marubatu_board.cファイルに作成します。

【marubatu_board.c】

```
/* ボードを表示する関数 */
void writeBoard(void) {
}

/* プレイヤーの選択した位置にマークをする関数
引数 str*      プレイヤーが指定したマーク位置文字列
戻り値    r    1：正常な値だったのでマーク完了
               0：不正な値だったのでマークできなかった
*/
int writeMark(char* str) {
    int r = 1;
    return r;
}
```

ヒント

*4：今の段階では、とりあえずmain関数をざっと見て、何をやっているかだいたいわかるようなプログラムを書くよう心がけましょう。

ヒント

*5：処理内容と戻り値、引数の詳細は、コメントを参照。

```
/* コンピュータの選択した位置にマークをする関数 */
void writeCompMark(void) {
}

/* マークが揃ったかどうかをチェックする関数
戻り値    r    1：マークが揃っている
               0：マークはまだ揃っていない
              -1：全てのボードが埋まったがどちらのマークもそろっていない
*/
int judge(void) {
    int r;
    return r;
}
```

各関数の引数と戻り値の説明を見れば、この段階でもmain関数の動作は全て理解できると思います。

(3) ヘッダーファイルの作成

marubatu.cの中で、marubatu_janken.cとmarubatu_board.cの中にある関数を使っているので、このままでは「各関数のプロトタイプ宣言がない！」と怒られてしまいます。2つの関数ファイルにあるプロトタイプ宣言を、marubatu_func.hに書きます。

あとひとつ、ボードの縦横のマスの数のマクロ[*6]を定義した、marubatu.hを用意します。

この2つのファイル「marubatu_func.h」「marubatu.h」を、marubatu.cの中でインクルードします。

○×ゲームは、合計5つのファイルで構成されることになります。

ヒント
*6：この時点のプログラムでは、まだマクロの値を使用していません。

● ○×ゲームを構成するファイル

- marubatu.c
- marubatu_janken.c
- marubatu_board.c
- marubatu_func.h
- marubatu.h

この時点[*7]で自作関数judgeと自作関数jankenの戻り値を1に固定してコンパイルを行うと、実行ファイルができあがります。

ヒント
*7：この時点では、自作関数writeMark、writeCompMark、writeBoardでは何もしていません。

```
C:¥source>marubatu.exe
```

```
【○×ゲーム】
場所を指定して下さい（例：a2） > a1

==================== ゲーム終了 =====================
プレイヤーの勝ち！
```

自作関数jankenでプレイヤーが○の状態にし、プレイヤーが1回だけ場所を入力した状態で「ボードが揃った」と判定して終了しているので、このような結果になります。

4 ジャンケン関数janken

では、関数の内容を肉づけしていきましょう。

この時限は、○×ゲームの序盤で使用する自作関数jankenとwriteBoardのみを作成します。

（1）関数の引数と戻り値

ジャンケンのときコンピュータが出す手はランダムです。よって、ジャンケンの勝敗の結果は、1回できまるとは限りません。あいこのときはもう一度ジャンケンをして、勝敗がつくまで繰り返します。自作関数jankenは、「1回だけジャンケンを行い、勝敗結果を出力する」処理を行います。関数の戻り値は、

●自作関数jankenの戻り値

```
1：プレイヤーの先攻
0：プレイヤーの後攻
-1：あいこでもう1回勝負
```

とします。引数はありません。

この関数を呼び出すには、main関数側で次のようにすると、

```
int p_turn; // プレイヤーターンの時は 1

do { p_turn = janken(); } while(p_turn < 0);
```

あいこの間はジャンケンを繰り返します。関数の戻り値をそのままプレイヤーのターンを表す変数p_turnに代入すれば、プレイヤーの先攻後攻が同時に設定できます。

（2）関数の処理

プレイヤーの手は入力、コンピュータの手はランダムです。第2日と第3日のプログラムを応用すれば、すぐに作成できます*8。最後にジャンケン勝負の結果を表示して、1、0、-1のどれかを返します。具体的な処理は、この2時限目の最初に掲載した実際のソースファイルの内容を参考にしてください。

ヒント

*8：使用する関数ごとに必要なヘッダーファイルも、ファイルの最初に記述しましょう。

(3) グローバル変数について

グローバル変数はひとつだけ用意しておく必要があります。先攻後攻がきまったときに、それを記録しておく変数です。

まず、marubatu.hに次のマクロを追加します。

```
#define MARU 1
#define BATU 2
```

marubatu_janken.cではmarubatu.hをインクルードし、次のようにグローバル変数を定義します*9。

```
int p_maru = BATU;  // プレイヤーが○の時はMARU、×の時はBATU
```

初期値はBATUです。関数の中でもしプレイヤーが先攻になったら、この値をMARUに変更します。

5 ボードを用意する

marubatu_board.cにボードに関する4つの関数を定義するため、ボードの状態を表す2次元配列をグローバル変数*10として用意します。

```
static int board[MTX][MTX];
```

この2次元配列に代入される値は、○か×を表す数値です。MARUかBATU（空白の場合は0）が入ります*11。

6 ボードを表示する自作関数writeBoard

ボードを表示する処理を担当している、自作関数writeBoardについて説明します。ボードの表示は、

```
   [1][2][3]
[a]       ×
[b] ○
[c]
```

このようになります。横のマスは1、2、3の数値で表します。縦のマスはa、b、cで表します。

表示の1行目「[1][2][3]」は簡単に実現できます。問題は縦の、

```
[a]
[b]
[c]
```

ヒント
*9：この変数p_maruは他のファイルから参照するので、static変数にしてはいけません。

ヒント
*10：他のファイルの関数からは使用しないので、staticをつけます。また、staticで宣言したint型配列は、すべて0で初期化されます。

ヒント
*11：マクロを利用しているため、marubatu_board.cからmarubatu.hをインクルードします。他に定義されている関数の中で必要になるヘッダーファイルは、その度にインクルードします。

という表示です。2次元配列のデータを端から順に参照する方法は、25ゲームを作ったときに理解したはずです。これは、2つのfor文を使って書きます。

縦方向は数値ではなく、アルファベットで考えます。アルファベット1文字は数値でも表すことができるので、それほど難しくありません。変数jを、初期値0から文字「a」[*12]にプラスした値を表示すれば、「a」「b」「c」となります。

```
for(j = 0; j < MTX; j++) {
    printf("[%c]", j+'a');
}
```

そして、残りのデータ部分も表示します。

2次元配列boardからひとつずつ取り出した値がBATUのときは×を表示し、MARUのときは○を表示します。0のときは空白です。全角半角が入り乱れるので、きちんと縦横の表示が揃うよう、スペースを利用して表示しましょう。また、この自作関数writeBoardは、引数も戻り値も必要ありません。

まとめ

2つの関数が完成した段階で実行すると、まずジャンケンを行って先攻後攻がきまります。今の段階では、「先攻者が1回入力（コンピュータの場合は自動決定）を行うだけで、先攻が勝つ」という結果に固定されています。

次の時限で残りの関数を作成して、ゲームを完成させましょう。

ヒント

＊12：'a'の部分は数値の97にしても同じ結果が得られます。

第10日 3時限目

【○×ゲームを作ろう③】○×ゲームを完成させよう

いよいよ本書の最後の時限です。これまで習得した知識を総動員して、○×ゲームプログラムを完成させましょう。

今回作成する例題

```
C:¥source>marubatu.exe
【○×ゲーム】
順番決めジャンケン・・・(グー:1 チョキ:2 パー:3を入力) > 1
コンピュータはパー！・・・プレイヤーが後攻（×）です
コンピュータは・・・b2
   [1][2][3]
[a]
[b]    ○
[c]
場所を指定して下さい（例：a2） > a1
   [1][2][3]
[a] ×
[b]    ○
[c]
コンピュータは・・・a3
   [1][2][3]
[a] ×     ○
[b]    ○
[c]
場所を指定して下さい（例：a2） > c1
   [1][2][3]
[a] ×     ○
[b]    ○
[c] ×
コンピュータは・・・b1
```

○×ゲームを行う

サンプルファイルはこちら　10days_c ▶ day10-03 ▶ marubatu.c

●このレッスンのねらい

2時限目の時点でゲーム序盤となる、ジャンケンでの先攻後攻ぎめとボードの表示を行う関数まで作成しました。
この時限は残りの関数を作成して、○×ゲームを完成させましょう。

プログラムを作成する

1 2時限目に作成したmarubatu.cとmarubatu_board.cに、新しく作成した部分を追加する

【marubatu.c】

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include "marubatu.h"
#include "marubatu_func.h"

extern int p_i,p_j; // 直前にマークしたマスのx,y座標

// ○×ゲーム
int main() {
(2時限目のmarubatu.cと同じなので略)

    for(;; p_turn = !p_turn) {
        if(p_turn){ // プレイヤーターン
            do {
                printf("場所を指定して下さい（例：a2） > ");
                scanf("%s", p_get);
            } while(!writeMark(p_get));

        } else {    // コンピュータターン
            writeCompMark();
            printf("コンピュータは・・・%c%d¥n", p_j+'a'-1, p_i);
        }
        writeBoard();
        r = judge();
        if(r != 0) { break;}
    }

(2時限目のmarubatu.cと同じなので略)

    return 0;
}
```

【marubatu_board.c】

```
#include <stdio.h>
#include <stdlib.h>
#include "marubatu.h"

static int board[MTX][MTX]; // ボード配列
int p_i,p_j; // 直前にマークしたマスのx,y座標
extern int p_maru; // プレイヤーが○の時はMARU、×の時はBATU

/* ボードを表示する関数 */
void writeBoard(void) {
(2時限目のmarubatu_board.cと同じなので略)
}

/* プレイヤーの選択した位置にマークをする関数
引数 str*      プレイヤーが指定したマーク位置文字列
戻り値   r     1：正常な値だったのでマーク完了
               0：不正な値だったのでマークできなかった
*/
int writeMark(char* str) {
    int r = 1;

    int i = str[1] - '0';
    int j = str[0] - 'a' + 1;
    if((i < 1 || i > MTX) || (j < 1 || j > MTX)) { r = 0; }
    else {
        if(board[j-1][i-1] == 0) {
            board[j-1][i-1] = p_maru;
            p_i = i; p_j = j;
        } else { r = 0; }
    }
    return r;
}

/* コンピュータの選択した位置にマークをする関数 */
void writeCompMark(void) {
    int count = 0;        // 空いているマスの数
    int c_maru = MARU;    // コンピュータのマーク
    int i,j;
    int r;
    int player = 0, computer = 0;    // プレイヤー，コンピュータクのマーク数
    int b = 0;   // ブランクの位置
```

```
int bmtx_i[100];
int bmtx_j[100];

if(p_maru == MARU) { c_maru = BATU; }

//横方向
for(j = 1; j <= MTX; j++) {
    //○の数と×の数のチェック
    for(i = 1, player = 0, computer = 0, b = 0; i <= MTX; i++) {
        if(board[j-1][i-1] == p_maru) { player++; }
        else if(board[j-1][i-1] == c_maru) { computer++; }
        else { b = i; }
    }
    //プレイヤーがリーチだった場合阻止する
    if((player+1 == MTX) && computer == 0) {
        board[j-1][b-1] = c_maru;
        p_i = b; p_j = j;
        return;
    }
}
//縦方向
player = 0; computer = 0; b = 0;
for(i = 1; i <= MTX; i++) {
    for(j = 1, player = 0, computer = 0, b = 0; j <= MTX; j++) {
        if(board[j-1][i-1] == p_maru) { player++; }
        else if(board[j-1][i-1] == c_maru) { computer++; }
        else { b = j; }
    }
    if((player+1 == MTX) && computer == 0) {
        board[b-1][i-1] = c_maru;
        p_i = i; p_j = b;
        return;
    }
}

//ナナメ
player = 0; computer = 0; b = 0;
for(i = 1; i <= MTX; i++) {
    if(board[i-1][i-1] == p_maru) { player++; }
    else if(board[i-1][i-1] == c_maru) { computer++; }
    else { b = i; }
}
if((player+1 == MTX) && computer == 0) {
```

```
            board[b-1][b-1] = c_maru;
            p_i = b; p_j = b;
            return;
        }
        player = 0; computer = 0; b = 0;
        for(j = 1; j <= MTX; j++) {
            if(board[j-1][MTX-j] == p_maru) { player++; }
            else if(board[j-1][MTX-j] == c_maru) { computer++; }
            else { b = j; }
        }
        if((player+1 == MTX) && computer == 0) {
            board[b-1][MTX-b] = c_maru;
            p_i = b; p_j = MTX-b+1;
            return;
        }

        //プレイヤーのリーチはなかったのでランダムにマークする
        for(j = 1; j <= MTX; j++) {
            for(i = 1; i <= MTX; i++) {
                if(board[j-1][i-1] == 0) {
                    bmtx_i[count] = i;
                    bmtx_j[count] = j;
                    count++;
                }
            }
        }
        r = rand()%count;
        board[bmtx_j[r]-1][bmtx_i[r]-1] = c_maru;
        p_i = bmtx_i[r]; p_j = bmtx_j[r];
}

/* マークが揃ったかどうかをチェックする関数
戻り値    r    1：マークが揃っている
               0：マークはまだ揃っていない
              -1：全てのボードが埋まったがどちらのマークも揃っていない
*/
int judge(void) {
        int i, j;
        int r;
        int j_maru = board[p_j-1][p_i-1] ;   //揃ったかどうか判定するマーク

        //横方向チェック
        for(i = 1, r = 1; i <= MTX; i++) {
            if(board[p_j-1][i-1] != j_maru) { r = 0; break; }
```

```
    }
    if(r) { return 1; }

    //縦方向チェック
    for(j = 1, r = 1; j <= MTX; j++) {
        if(board[j-1][p_i-1] != j_maru) { r = 0; break; }
    }
    if(r) { return 1; }

    //ナナメチェック
    if(p_j == p_i) {
        for(i = 1, r = 1; i <= MTX; i++) {
            if(board[i-1][i-1] != j_maru) { r = 0; break; }
        }
    }
    if(r) { return 1; }

    if(p_j+p_i == MTX+1) {
        for(i = 1, r = 1; i <= MTX; i++) {
            if(board[i-1][MTX-i] != j_maru) { r = 0; break; }
        }
    }
    if(r) { return 1; }

    //あいこ
    for(j = 1, r = -1; j <= MTX; j++) {
        for(i = 1; i <= MTX; i++) {
            if(board[j-1][i-1] == 0) { return 0; }
        }
    }
    return r;
}
```

2 **入力できたらそれぞれの名前**[*1]**(手順①の【】内参照)で、「C:¥source」ディレクトリ下に保存する**

ヒント
*1：拡張子に注意して保存しましょう。

3 **コマンドプロンプトを起動し、「C:¥source」ディレクトリに移動して、ソースファイルを3つともコンパイルする**

```
C:¥Users¥user>cd ¥source

C:¥source>gcc -o marubatu marubatu_janken.c marubatu_board.c marubatu.c
```

4 プログラムを実行する。マークの入力とボードの表示、マークが揃ったかどうか判定されれば成功！

```
C:¥source>marubatu.exe
```

```
【○×ゲーム】
順番決めジャンケン・・・( グー :1  チョキ :2  パー :3 を入力 ) >
コンピュータはパー！・・・プレイヤーが先攻（○）です
   [1][2][3]
[a]
[b]
[c]
場所を指定して下さい（例：a2） > a1
   [1][2][3]
[a] ○
[b]
[c]
コンピュータは・・・a2
   [1][2][3]
[a] ○ ×
[b]
[c]
場所を指定して下さい（例：a2） > b2
   [1][2][3]
[a] ○ ×
[b]    ○
[c]
コンピュータは・・・c3
   [1][2][3]
[a] ○ ×
[b]    ○
[c]       ×
場所を指定して下さい（例：a2） > b1
   [1][2][3]
[a] ○ ×
[b] ○ ○
[c]       ×
コンピュータは・・・b3
   [1][2][3]
[a] ○ ×
[b] ○ ○ ×
```

```
[c]        ×
場所を指定して下さい（例：a2） > c1
   [1][2][3]
[a] ○  ×
[b] ○  ○  ×
[c] ○     ×

==================== ゲーム終了 =====================
プレイヤーの勝ち！
```

解説

1 プレイヤーが選択した位置にマークする自作関数writeMark

プレイヤーは、自分がマークする位置をキーボードで入力します。マークできない位置を指定することもありますので、入力値が不正な場合はmain関数の中で再度入力を促します。

```
do {
     printf("場所を指定して下さい（例：a2） > ");
     scanf("%s", p_get);
} while(!writeMark(p_get));
```

自作関数writeMarkは、プレイヤーからの入力値を引数に、戻り値として次の2種類を返すようにします。

●自作関数writeMarkの戻り値

1：正常な値だったのでマーク完了
0：不正な値だったのでマークできなかった

プレイヤーがマークしたい場所の入力は、x方向とy方向の両方を、一度に書いて指定します。例えば、下の図の位置の場合は

```
   [1][2][3]
[a] ○
[b]
[c]
```

「a1」と表現するので、y方向の「a」とx方向の「1」をつなげて書きます。他には、例えば「c3」や「b1」などと入力します。アルファベットと数字の組みあわせなので、これは文字列の扱いになります。

文字列であるマーク位置を自作関数writeMarkに引数として渡すには、受け取り側でポ

インタを利用します。

```
int writeMark(char* str) {
```

と受け取った場合、「a2」が入力されてきたら、str[0]に最初の文字であるy方向の位置「a」が入り、str[1]に次の文字であるx方向の位置「2」が入ることになります。これを、2次元配列のx方向とy方向それぞれの位置に変換します。x方向が2となっていればそのまま数値の2としたいところですが、文字型ということは文字の「2」なので、数値に変換しなければなりません。y方向のa、b、cも、同じく「1」「2」「3」に対応変換します。

```
int i = str[1] - '0';
int j = str[0] - 'a' + 1;
```

変換した値の位置が3×3のボードから飛び出していたら不正な値*2です。ボードの中におさまっていても、すでにその位置のマスに値が入っていたら、それも不正な値です。まだそのマスに値が入っていなければ、marubatu_janken.cに定義した、

ヒント
*2：マクロMTXの最大値は9です。実際のゲームとしては無謀ですが、9×9までの○×ゲームを作ることが可能です。

```
int p_maru;
```

の値を入れます。プレイヤーが○のときは、p_maruにはMARU、つまり1が入っているので、これをそのまま設定します。プレイヤーが×のときは、p_maruにはBATU、つまり2が入っています。

よって、この変数p_maruは、marubatu_board.cファイルの中でexternする必要があります。

```
int p_i,p_j; // 直前にマークしたマスのx,y座標
extern int p_maru; // プレイヤーが○の時はMARU、×の時はBATU
```

このときマークした配列boardのx方向とy方向の位置*3（数値の1、2、3）を、それぞれ変数p_i、変数p_jに入れておきます。これもグローバル変数として定義します。

ヒント
*3：プレイヤーの入力にsscanf関数を使い、最初からy座標の文字とx座標の数値に分ける方法もあります。この場合、x座標の数値は、入力値をそのまま使うことができます。

② コンピュータの選択した位置にマークする自作関数writeCompMark

今度は、コンピュータがマークします。コンピュータの選択するボードの位置は不正な値にならないはずなので、この関数を呼び出すのは1回だけです。関数内で位置を決定してboardへの設定も行うので、この関数には引数も戻り値も必要ありません。

さらっと書きましたが、実はこの関数、非常にやっかいです。空いている場所にランダムにマークするなら簡単ですが、それではほとんどプレイヤーの勝ちになってしまいます。コンピュータの勝つ確率を上げるような工夫が必要なのです。

(1) 思考するコンピュータ

人間同士でゲームをすると、自分があとひとつで1列揃う位置を選びます。

```
    [1][2][3]
[a] ○  ×
[b] ○  ×
[c]
```

この状態だったら、○は次に「c1」の位置にマークすれば、勝利します。しかし、的外れの「a3」の位置にマークしたら、相手の×は、次に「c2」の位置にマークすれば勝利します。

```
    [1][2][3]
[a] ○  ×  ○
[b] ○  ×
[c]    ×
```

○を持った方は、たとえ自分がリーチであることを見逃しても、相手がリーチ状態*4にあったらそれを阻止する場所、つまりc2の位置にマークしていれば少なくともすぐ負けることはありません。

```
    [1][2][3]
[a] ○  ×
[b] ○  ×
[c]    ○
```

思考を持たないコンピュータとある程度対等にゲームを行うためには、最低限「コンピュータの負けを阻止する」機能があればよいでしょう。

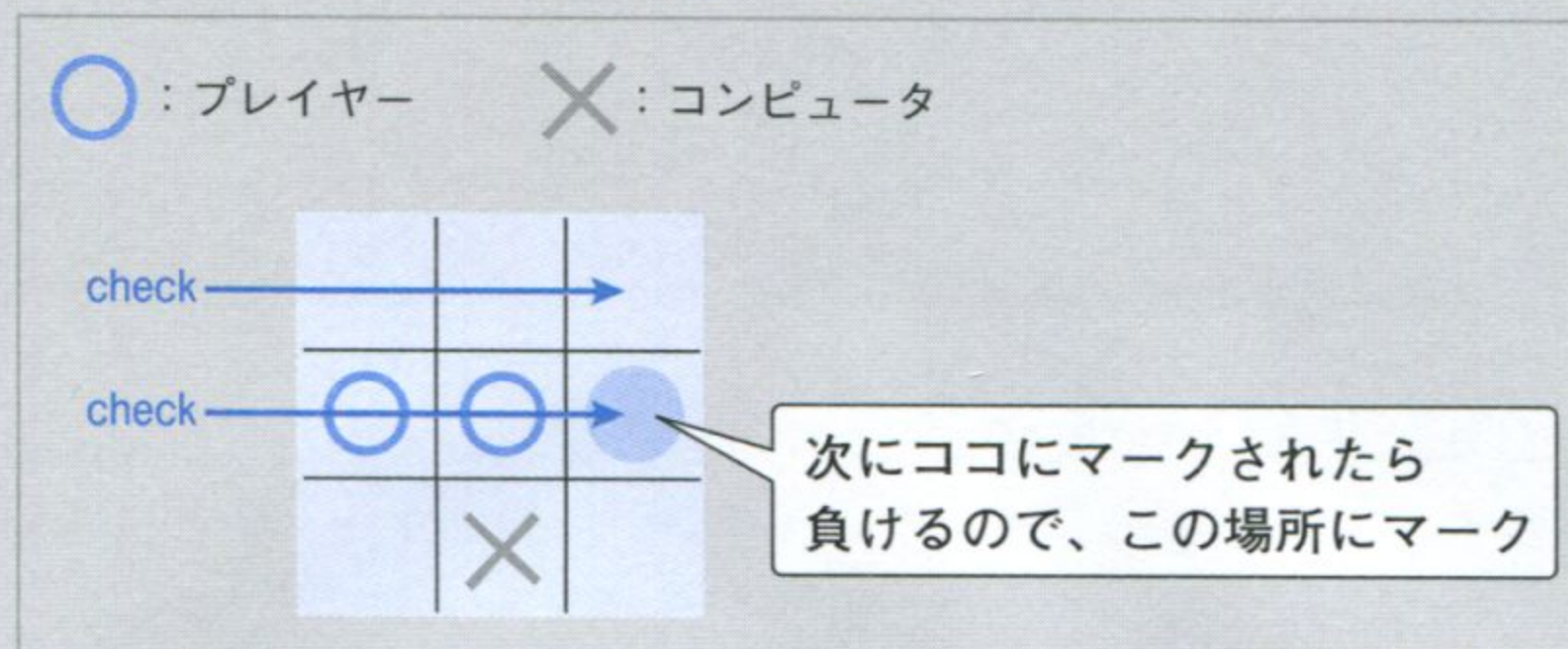
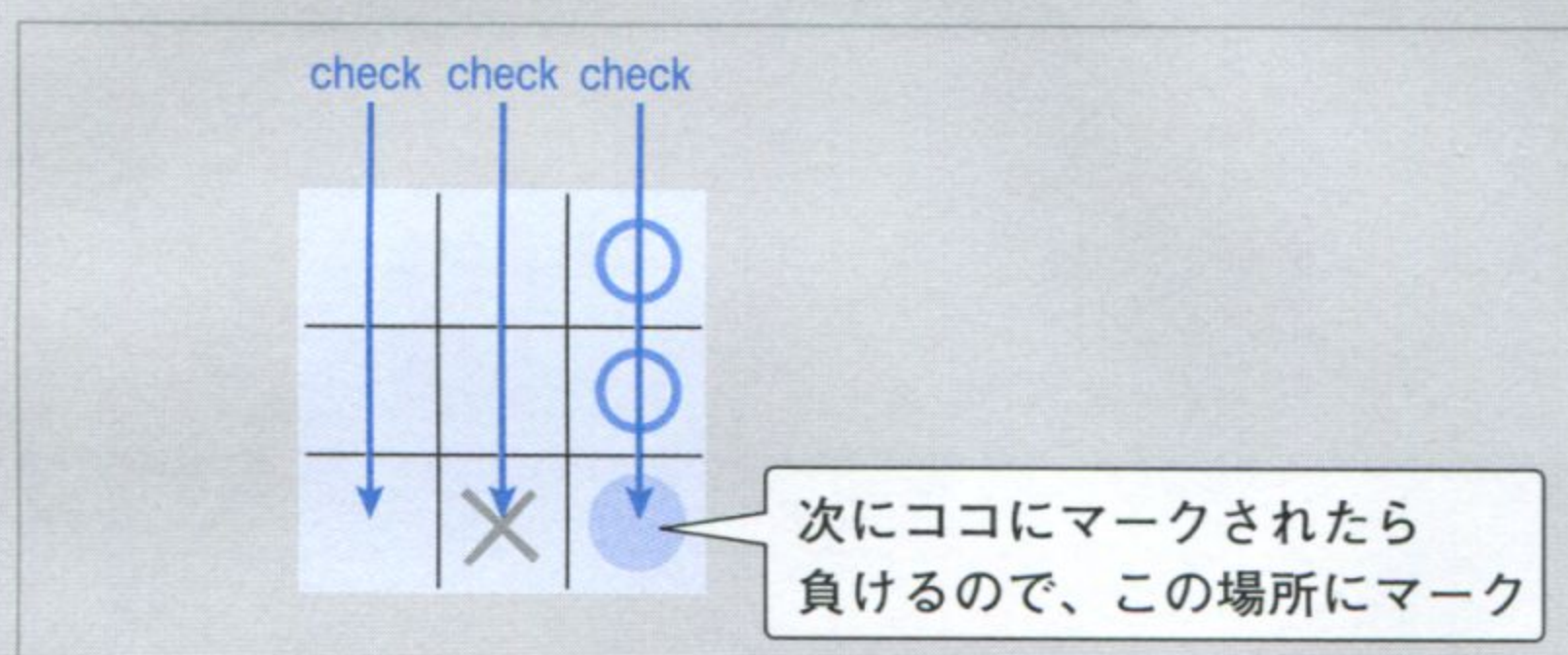
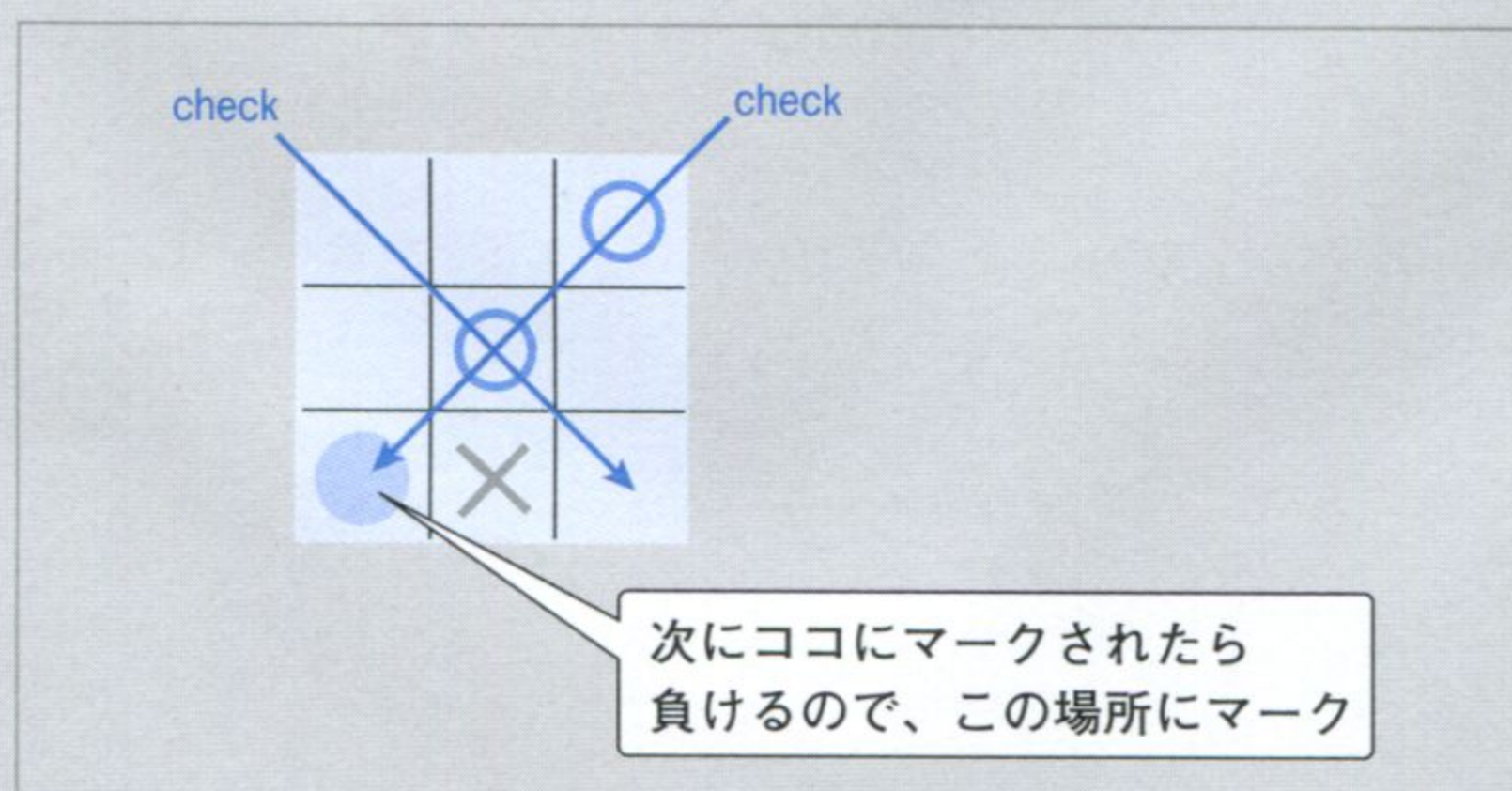
(2)「負けを阻止する」機能

自作関数writeCompMarkでは、次の3つさえチェックしてマークすれば、まずあっさりと負けることはありません。

ヒント

*4：あとひとつで1列揃う状態を、リーチ状態と呼びます。

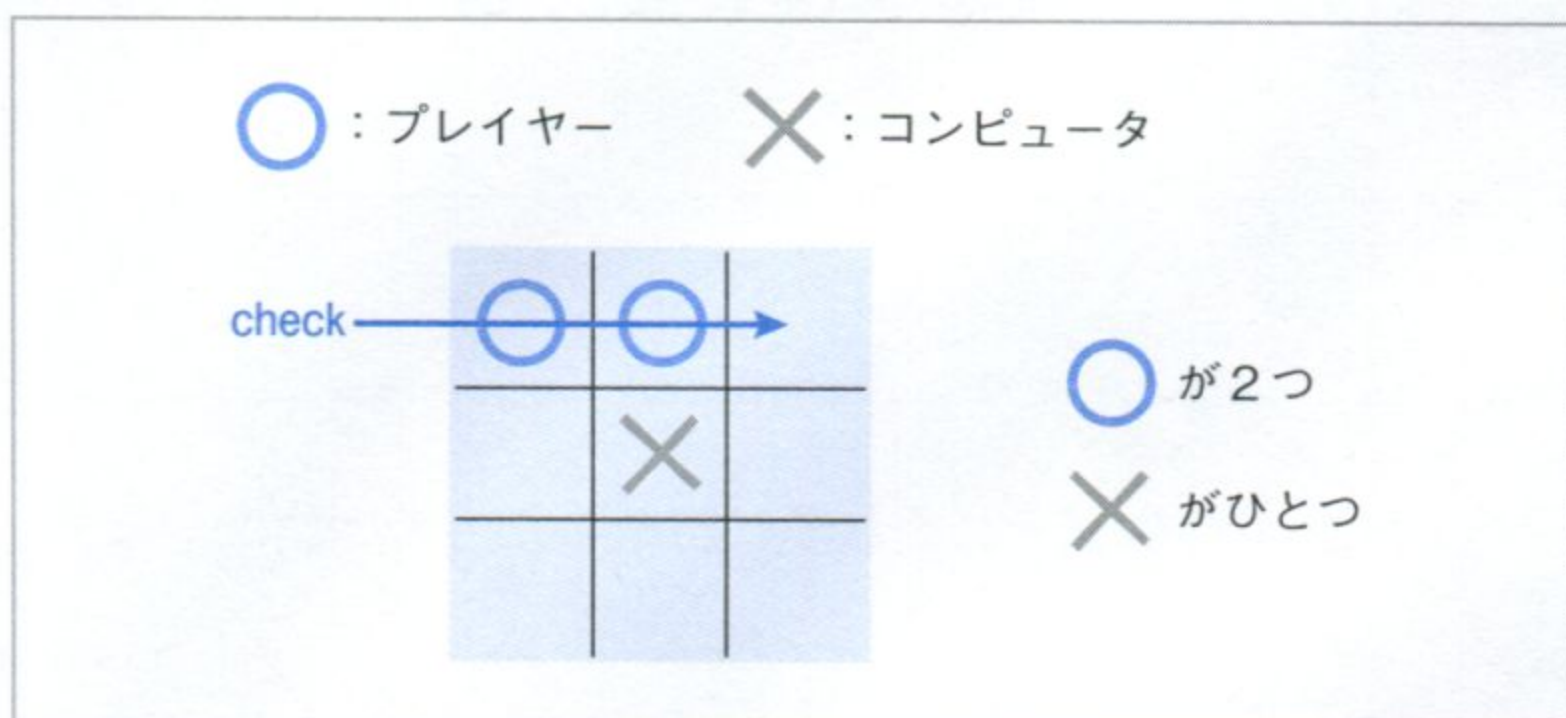
●負けを阻止するためのポイント

・横方向チェック……y列を1列ずつチェックして、プレイヤーがリーチ状態の列を見つけたら、そこにマーク

・縦方向チェック……x列を1列ずつチェックして、プレイヤーがリーチ状態の列を見つけたら、そこにマーク

・ナナメチェック……ナナメの2列をチェックして、プレイヤーがリーチ状態の列を見つけたらそこにマーク

そして、どのチェックにもひっかからなければ、縦横ランダムな位置にマークをする機能をつけましょう。次に、この横・縦・ナナメチェック処理の方法を考えます。

(3) 横方向にチェックする処理を考える

すべて説明すると長くなるので、横方向のチェックのみ、作り方を説明します。プレイヤーのマークが○、コンピュータが×の場合を例にします。

まず、横方向に1列ずつチェックします。1列のうち、プレイヤーのマークである○を表すMARUが入っているマスの数と、コンピュータのマークである×を表すBATUが入っているマスの数をカウントしておきます。空いているマスの位置もおぼえておきましょう。

●横方向のチェック

1列数えおわったときに、もし○が2つで×がひとつもなければ、プレイヤー側のリーチ状態です。よってコンピュータは、この状態のときは、先ほどおぼえた空いている位置にマークします[*5]。決定したら、自作関数writeCompMarkはreturnを使って出てしまいましょう。

またマーク位置がきまったら、その位置を自作関数writeMarkでも使ったグローバル変数p_i、p_jにセットします。p_i、p_jは、今マークした場所のx座標、y座標を表します。

ヒント

*5：変数bの位置は、ひとつだけ空いている状態だったら、その位置が記録されています。リーチ状態になければ、この位置は0か、最後に空白だった場所が記録されていますが、処理には関係ありません。

```
int c_maru = MARU;   // コンピュータのマーク
if(p_maru == MARU) { c_maru = BATU; }

for(j = 1; j <= MTX; j++) {
    // ○の数と×の数のチェック
    for(i = 1, player = 0, computer = 0, b = 0; i <= MTX; i++) {
        if(board[j-1][i-1] == p_maru) { player++; }
        else if(board[j-1][i-1] == c_maru) { computer++; }
        else { b = i; }
    }
    // プレイヤーがリーチだった場合阻止する
    if((player+1 == MTX) && computer == 0) {
        board[j-1][b-1] = c_maru;
        p_i = b; p_j = j;
        return;
    }
}
```

もし横方向にリーチ状態がなかったら、次は縦方向のチェックを行います。縦方向のチェック方法は、これとほぼ同様に作れます。同じようにマークする位置が決定したら関数を抜け、決定しなければナナメチェックを実行します。

(4) リーチ状態のチェックにひっかからない場合の処理について考える

どのリーチチェックにもひっかからなかった場合は、空いている位置をランダムに選択してマーク位置を決定します。まず、配列boardのすべてのマスを順にチェックします。空いてる位置が見つかったら、そのx座標、y座標をそれぞれ1次元配列に格納していきます。

```
int bmtx_i[100];
int bmtx_j[100];
```

これは、チェック可能な位置のx座標、y座標をそれぞれ順に格納したものです。つまり、配列boardがこの図の状態のとき、

```
    [1][2][3]
[a] ○
[b] ×  ×  ○
[c]    ○
```

最初の空白は、xが2、yが1の位置です（位置は1から数える)。よって、bmtx_i[0]は2、bmtx_j[0]は1となります。次の空白は、xは3、yは1です。これを配列に追加します。

```
bmtx_i[] = { 2, 3 }
bmtx_j[] = { 1, 1 }
```

最後のマスまでチェックしおわると、空白のマスの位置をそれぞれ記録した配列は、次のようになります。

```
bmtx_i[] = { 2, 3, 1, 3 }
bmtx_j[] = { 1, 1, 3, 3 }
```

このあと、rand関数の値を、空白のマスの数をカウントした変数countで割った数字を算出します。この値が2だったとき、

```
bmtx_i[] = { 2, 3, 1, 3 }
bmtx_j[] = { 1, 1, 3, 3 }
                  ↑
```

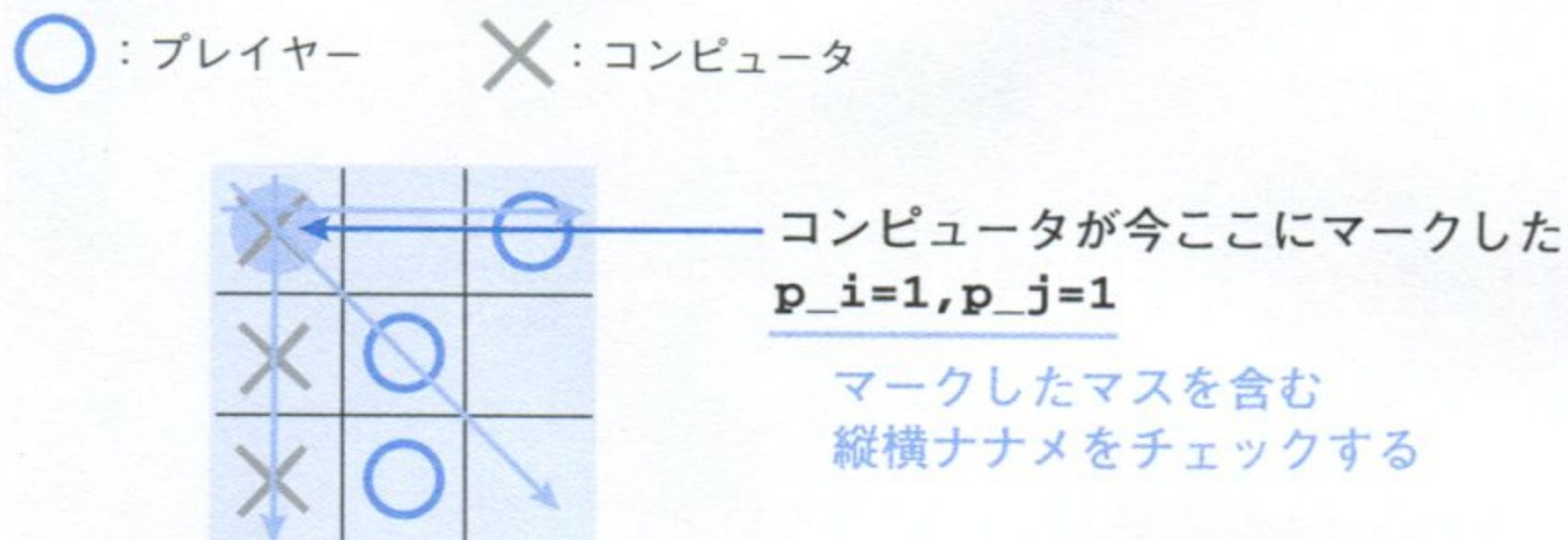
つまりy=3、x=1をマークする位置に決定します。

```
r = rand()%count;
board[bmtx_j[r]-1][bmtx_i[r]-1] = m_maru;
p_i = bmtx_i[r]; p_j = bmtx_j[r];
```

最後にマークがきまったら、その位置をグローバル変数p_i、p_jにセットしておきましょう。

3 マークがそろったかどうかをチェックする自作関数judge

プレイヤー、コンピュータのどちらの場合もマークが終了したら、今ので1列揃ったかどうかをチェックします。直前にマークした位置は、必ずmarubatu_board.cのグローバル変数p_i、p_jに格納されています。よって、このマスを含む縦と横とナナメ方向を見て、どれか1列すべてマークが揃っているかどうかをチェックします。

●揃ったかどうかをチェック

この関数に引数は必要ありませんが、戻り値は、次の3種類を返します。

●自作関数judgeの戻り値

1：マークが揃っている
0：マークはまだ揃っていない
-1：全てのボードが埋まったがどちらのマークも揃っていない

自作関数writeCompMarkでの考え方を利用すればすぐできるので、横方向のソースのみ紹介します。まず、先ほどマークしたのは何かを確認します。

```
int j_maru = board[p_j-1][p_i-1] ;
```

マークが1であった場合は○なので、先ほどマークしたマスの位置の横1列に、すべて同じ1が設定されているかどうかチェックします。

```
for(i = 1, r = 1; i <= MTX; i++) {
    if(board[p_j-1][i-1] != j_maru) { r = 0; break; }
}
if(r) { return 1; }
```

すべて同じマークだったら1を返して、関数を終了します。ひとつでも違うマークが出たらその場で横方向のチェックを終了して、次の縦方向のチェックに入ります。

縦横ナナメのチェックが終了した段階でどこもマークが揃っていなければ、引き分けのときもあります。配列boardのすべてのマスをチェックして、すべてが1か2で埋まっていたら、空いているマスはありません。よって、これは引き分けとなるので-1を返します。

これで、すべての関数が揃いました。

4 外部参照変数、ローカル変数ついて考える

外部参照変数について確認しましょう。

marubatu_janken.cの中で、プレイヤーが○か×のどちらなのかは、変数p_maruを使って定義しています。

次にmarubatu_board.cを確認します。変数p_maruはこのファイルの中の関数でよく使うので、externして使いましょう。

ゲームのボードを表す2次元配列変数boardは、このmarubatu_board.c内のみで使うのでstaticで宣言します。

次の変数p_i、p_jは、どちらのターンでも直前にマークした位置のx座標、y座標です。これは、実はmarubatu.cで使っているので、staticをつけないでおきます。

コンピュータがマークした位置を表示する機能は、marubatu_board.cに作成した自作関数writeCompMarkには入っていません。よって、main関数のmarubatu.cでmarubatu_board.cの変数p_i、p_jをexternして使います。

```
extern int p_i,p_j;

int main() {
        (略)
    printf("コンピュータは・・・%c%d¥n", p_j+'a'-1, p_i);
```

コンピュータがマークした位置を表示する処理も自作関数writeCompMarkに入れてしまえばよさそうですが、この関数ではボード配列の縦・横・ナナメをチェックして、マークすべき場所が見つかり次第マークを行い、関数を抜けています。すると、このprintf関数を何箇所にも入れなくてはならないので、今回はmain関数の中で書くようにしています。

5 ○×ゲームプログラムを完成させる

最後に、main関数の中で勝敗結果を出力する部分を説明します。勝敗結果は、コンピュータ側のマークが揃うか、プレイヤー側のマークが揃うか、マークが揃うことなくすべてのマスが埋まってしまう引き分け、の3種類です。

判定には、marubatu_board.cの中にある自作関数judgeを使い、この結果を変数rに格納します。変数rが-1か1のときは勝敗が判定しているので、繰り返しを終了します。

rが-1のときは引き分けです。そのとおり出力します。繰り返し終了時にプレイヤーのターンだったときは、プレイヤーの勝利です。逆にコンピュータのターンのときは、プレイヤーの負けになります。

この判定部分をmain関数に入れて、○×ゲームプログラムを完成させます。

まとめ

長いプログラムほど、改良の余地はあります。作ったプログラムが問題なく動いたら、次は「機能的に追加する部分はないか？」「プログラムに無駄な部分はないか？」を考えてみるようにしましょう。この作業の積み重ねにより、最初から無駄のないプログラムを素早く書くことができるようになります。

練習問題

Q

自作関数writeCompMarkの中からリーチ状態をチェックする部分を抜き出し、関数化しなさい。その際、コンピュータ側のリーチ状態も見つけて先にマークする機能を追加すること。また、marubatu.cの中で外部変数p_i、p_jをexternしなくてもよいようにすること。

●関数仕様

```
/*リーチ状態の場所にマークをする関数
引数      1：コンピュータリーチチェック
          2：プレイヤーリーチチェック
戻り値    1：マークを行った
          0：マークをしていない
*/
int compMarkCheck(int mode);
```

解答は巻末に

本書のまとめ

本書では、C言語の基本を学習してきました。本書で紹介できなかった機能も多く存在しますが、ここから先は、基本をしっかりマスターしてから学ぶべき内容です。

本書をここまで読み進めてきた方なら、今現在身についていることだけでも十分なプログラムが組めるはずです。あとは、このまま中級者向けC言語を学習してもよいし、C++言語や、Windowsアプリケーションを作るためにVisual C++に挑戦してみるのもよいでしょう。

どちらに進むにしても少しだけ大変になりますが、C言語の基本が身についていれば、怖いことはありません。

誰でも最初は初心者です。いろいろなものに挑戦してみてください。

第 ! 日

付録

練習問題の解答

練習問題の解答

1
1

第１日
１時限目

A 解答プログラムはこちら　10days_c ▶ day01-01 ▶ sample_renshuu.c

sample.cの４行目を変更する。

【sample_renshuu.c】

```
#include <stdio.h>

int main() {
    printf("C 言語の勉強をはじめるよ！ ");
    return 0;
}
```

2
1

第２日
１時限目

A 解答プログラムはこちら　10days_c ▶ day02-01 ▶ 2-1_renshuu.c

【2-1_renshuu.c】

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    int d;

    srand(time(NULL));
    d = rand()%6 + 12;
    printf("%d", d);
    return 0;
}
```

2
2

第2日
2時限目

A 解答プログラムはこちら 10days_c ▶ day02-02 ▶ 2-2_renshuu.c

【2-2_renshuu.c】

```
#include <stdio.h>

int main() {
    double d;    // 実数を格納する変数
    char str[10]; //10-1 文字以内の文字列を格納する変数

    printf(" 実数を入力してください > ");
    scanf("%lf", &d);
    printf(" 文字列を入力してください > ");
    scanf("%s", str);

    printf(" 文字列 %10s ：実数 %.3lf", str, d);
    return 0;
}
```

2
3

第2日
3時限目

A 解答プログラムはこちら 10days_c ▶ day02-03 ▶ 2-3_renshuu.c

【2-3_renshuu.cより抜粋】

```
    (略)
    printf(" コンピュータは ");
    switch (computer) {
        case 1: printf(" グー ");
                break;
        case 2: printf(" チョキ ");
                break;
        case 3: printf(" パー ");
                break;
        default:    break;
    }
    printf(" ！ ");
    return 0;
}
```

2/4 第2日 4時限目

A 解答プログラムはこちら 10days_c ▶ day02-04 ▶ janken_renshuu.c

「勝敗の判定と結果表示」部分のif～else文の最後にelse文を追加する。

【janken_renshuu.cより抜粋】

```
    (略)
    } else if(player == 3) {
        if(computer == 1) { printf("プレイヤーの勝ち¥n"); }
        else { printf("コンピュータの勝ち¥n"); }
    } else {
        printf("1,2,3のどれかを入力してね！¥n");
    }
    return 0;
}
```

3/2 第3日 2時限目

A 解答プログラムはこちら 10days_c ▶ day03-02 ▶ 3-2_renshuu.c

【3-2_renshuu.cより抜粋】

```
    (略)
        if(computer == player){
            printf("あいこ¥n");
        } else if((player == 1 && computer == 2) ||
                  (player == 2 && computer == 3) ||
                  (player == 3 && computer == 1)) {
            printf("プレイヤーの勝ち¥n");
        } else {
            printf("コンピュータの勝ち¥n");
        }
    (略)
```

3/4 第3日 4時限目

A 解答プログラムはこちら 10days_c ▶ day03-04 ▶ yakyuuken_renshuu.c

変数jを変数宣言に追加し、scanf関数のあとのwhile文にj++;処理を入れ、その直後に「j>0」ならplayerの入力値を0にし、jの値を0に戻す処理を追加する。

【yakyuuken_renshuu.cより抜粋】

```
    (略)
    int i;
    int j = 0;

    (略)
        do {
            printf("よよいのよい！¥n");
            printf("(グー:1 チョキ:2 パー:3を入力) > ");
            player = 0;
            scanf("%d", &player);
            while (getchar() != '¥n') { j++; }
            if(j > 0) { player = 0; j = 0; }
            computer = rand()%3 + 1;
```

4/1 第4日 1時限目

A 解答プログラムはこちら 10days_c ▶ day04-01 ▶ 4-1_renshuu.c

【4-1_renshuu.c】

```
#include <stdio.h>

int main() {
    int kotae[10];  // 解答を入れる配列
    int i;

    for(i = 0; i < 10; i++) {
        kotae[i] = 0;
        printf("値を入力してください > ");
        scanf("%d", &kotae[i]);
        while (getchar() != '¥n') {  }
    }
    for(i = 0; i < 10; i++) {
        printf("%d 回目の入力値：%d¥n", i+1, kotae[i]);
    }

    return 0;
}
```

4
2
第4日
2時限目

A 解答プログラムはこちら 10days_c ▶ day04-02 ▶ noutore1_renshuu.c

【noutore1_renshuu.cより抜粋】

```
(略)
for(i = 0; i < 5; i++) {
    r = -1;
    j = rand()%19 + 2;
    k = rand()%8 + 2;
    printf("%2d × %2d = ", j, k);
    start = clock();
    scanf("%d", &r);
    while (getchar() != '¥n') {  }
    end = clock();
    kotae[i] = r;
    seikai[i] = j * k;
    jikan[i] = (double)(end-start) / CLOCKS_PER_SEC;
}
for(i = 5; i < 10; i++) {
    r = -1
    j = rand()%19 + 2;
    k = rand()%8 + 2;
    printf("%2d ÷ %2d = ", j*k, j);
    start = clock();
    scanf("%d", &r);
    while (getchar() != '¥n') {  }
    end = clock();
    kotae[i] = r;
    seikai[i] = k;
    jikan[i] = (double)(end-start) / CLOCKS_PER_SEC;
}
(略)
```

4
3
第4日
3時限目

A 解答プログラムはこちら 10days_c ▶ day04-03 ▶ 4-3_renshuu.c

【4-3_renshuu.c】

```
#include <stdio.h>
#include <string.h>

int main() {
    int i, j, str_len;
    char input_str[20];
    char output_str[20];

    printf("文字列を入力してください > ");
    scanf("%s", input_str);
    str_len = strlen(input_str);

    for(i = str_len-1, j = 0; i >= 0; i--, j++) {
        output_str[j] = input_str[i];
    }
    output_str[j] = '¥0';
    printf("反転文字列：%s", output_str);

    return 0;
}
```

5
3
第5日
3時限目

A 解答プログラムはこちら 10days_c ▶ day05-03 ▶ yamanotesen_renshu.c

条件さえ満たすプログラムであれば作り方は自由です。
解答プログラムは参考にどうぞ。ここでは、コンピュータかプレイヤーが選んだデータを、配列から抜いていく方法をとっています。

"1", "2", "3", "4", "5"
↓ ("3"が選ばれた)
"1", "2", "4", "5"

【yamanotesen_renshu.c】

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

int main() {
    char *data[] = { "おひつじ", "おうし", "ふたご", "かに",
                     "しし", "おとめ", "てんびん", "さそり",
                     "いて", "やぎ", "みずがめ", "うお" };
    int data_len = 12;  //データの数
    char input[10];     //プレイヤー入力値
    int i, j, p;
    int check;
    int p_turn = 0; //プレイヤーのターンの場合は1

    srand(time(NULL));

    printf("古今東西山手線ゲ～～ム！¥n");
    printf("お題：星座の名前¥n");
    for(i = 1; i <= data_len; i++, p_turn = !p_turn) {
        Sleep(500);
        if(p_turn == 0) { printf("コンピュータ"); }
        else { printf(" プレイヤー"); }
        printf("の番  ちゃん  ");
        Sleep(500);
        printf("ちゃん！ > ");
```

```
        if(p_turn == 0) {
            p = rand()%(data_len-(i-1));
            printf("%s¥n", *(data+p));
        } else {
            scanf("%s", input);
            while (getchar() != '¥n') {  }
            check = 0;
            for(j = 0; j < data_len-(i-1); j++) {
                if(strcmp(*(data+j), input) == 0) {
                    check = 1;
                    p = j;
                    break;
                }
            }
            if(check == 0) { break; }
        }
        for(j = p; j < data_len-i; j++) {
            data[j] = data[j+1];
        }
    }

    if(check == 1) {
        printf("あなたの勝ち！");
    } else {
        printf("あなたの負け！");
    }
    return 0;
}
```

6
3
第6日
3時限目

A 解答プログラムはこちら 10days_c ▶ day06-03 ▶ itudoko_data_renshuu.c

【itudoko_data_renshuu.c】

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    FILE *fp; // 入力ファイルのファイルポインタ
    char datafile[] = "itudoko.txt";
    int k;
    char str[256];
    char *kubun[] = { "serifu", "hito", "shita", "shurui" };
    char input_str[512];
    char *data[1000];     // データ格納配列
    char buff[512];       // ファイル読み込みバッファ
    int data_c = 0;       // データ数
    int i;
    int check;

    // 入力ファイルをオープンする
    if((fp = fopen(datafile, "a+")) == NULL) {
        printf("ファイルオープンエラー¥n");
        exit(1);
    }

    fseek(fp, 0, SEEK_SET);
    while(fgets(buff, sizeof buff, fp) != NULL){
        data[data_c] = (char*)malloc(strlen(buff) + 1);
        strcpy(data[data_c++], buff);
    }

    while(1) {
            printf("1～5までの数値を入力してください¥n");
            continue;
        }
        printf("内容は? > ");
        fgets(str, 256, stdin);
        if(str[0] == '¥n') {
            printf("内容を入力してください¥n");
            continue;
```

```
        }
        if((strlen(str) == 255) && (str[254] != '¥n')) {
            printf("文字数オーバーです¥n");
            while (getchar() != '¥n') { }
            continue;
        }

        sprintf(input_str, "%s¥t%s", kubun[k-1], str);
        check = 0;
        for(i = 0; i < data_c; i++) {
            if(strcmp(data[i], input_str) == 0) {
                check = 1;
                break;
            }
        }
        if(check == 1) { printf("データはすでに存在します¥n"); continue; }
        fprintf(fp, "%s", input_str);
        data[data_c++] = input_str;
    }

    fclose(fp);
    for(i = 0; i < data_c; i++) { free(data[i]); }
    return 0;
}
```

7
1
第7日
1時限目

A 解答プログラムはこちら 10days_c ▶ day07-01 ▶ 7-1_renshuu.c

【7-1_renshuu.c】

```
#include <stdio.h>
#include <stdlib.h>

int func1(int d1, int d2) {
    int total = d1 + d2;
    return total;
}

int main(int argc, char* argv[]) {
    int d = 0;
    int d1 = 0, d2 = 0;

    if(argc == 3) {
        d1 = atoi(argv[1]);
        d2 = atoi(argv[2]);
        d = func1(d1, d2);
        printf("%d + %d = %d¥n", d1, d2, d);
    }
    return 0;
}
```

7/4
第7日
4時限目

A 解答プログラムはこちら　10days_c ▶ day07-04 ▶ blackjack_renshuu.c

【blackjack_renshuu.c】

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int card[4][13];
char total[13] = { 11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10 };
char *digit[13] = { "A", "2", "3", "4", "5", "6",
                    "7", "8", "9", "10", "J", "Q", "K" };
char *mark[4] = {
    "ハート",
    "ダイヤ",
    "スペード",
    "クローバー"
};

int drawCard(int h);

int main() {
    int dealer; //ディーラーの引いたカードの合計
    int player; //プレイヤーの引いたカードの合計
    int draw_digit; //引いたカードの数
    int draw_mark; //引いたカードのマーク
    char y_n; //カードを引くか引かないかの答え

    srand(time(NULL));
    printf("【ブラックジャック】¥n");

    //ディーラーが引く
    printf("ディーラー一枚目：");
    dealer = drawCard(2);
    printf("他は伏せる¥n¥n");

    //プレイヤーが引く
    printf("プレイヤー一枚目：");
    player = drawCard(1);
    printf("プレイヤー二枚目：");
    player += drawCard(1);
```

```
    //ディーラー2枚目以降
    do{
        dealer += drawCard(0);
    } while(dealer <= 16);

    //プレイヤー3枚目以降
    while(player < 21) {
        printf("もう1枚引きますか?(y/n) > ");
        scanf("%c" , &y_n);
        while (getchar() != '¥n') {  }
        if(y_n == 'y') {
            player += drawCard(1);
        } else if (y_n == 'n') { break; }
    }

    printf("¥nディーラー：%d点  プレイヤー：%d点¥n", dealer, player);

    //勝敗の判定
    if ((dealer <= 21 && player > 21)
                || (dealer <= 21 && dealer > player)) {
        printf("ディーラーの勝ち！¥n");
    } else if ((player <= 21 && dealer > 21)
                || (player <= 21 && player > dealer)) {
        printf("プレイヤーの勝ち！¥n");
    } else {
        printf("引き分け¥n");
    }
    return 0;
}

/* カードを引く関数
引数 h：カード内容表示フラグ
        プレイヤーターンの時は1、ディーラー2枚目以降の時は0、
        ディーラー1枚目のときは2
戻り値 r：引いたカードの点数
*/
int drawCard(int h) {
    int draw_mark, draw_digit; //引いたカードのマークと数
    int r; //引いたカードの点数
    char y_n; //カードを11として計算するかどうかの答え

    do {
        draw_mark = rand() % 4 + 1;
```

```
        draw_digit = rand() % 13 + 1;
    } while (card[draw_mark-1][draw_digit-1]);
    card[draw_mark-1][draw_digit-1] = 1;
    if(h > 0) {
        printf("%sの%s¥n", mark[draw_mark-1], digit[draw_digit-1]);
    }

    if((h == 1) && draw_digit == 1) {
        // プレイヤーターンで引いた数が1のとき
        do {
            printf("11として計算しますか? (y/n) > ");
            scanf("%c" , &y_n);
            while (getchar() != '¥n') {  }
            if(y_n == 'y') { r = total[draw_digit-1];  }
            else if (y_n == 'n') { r = 1;  }
        } while(!(y_n == 'y' || y_n == 'n'));
    } else { r = total[draw_digit-1]; }
    return r;
}
```

8-3 第8日 3時限目

A 解答プログラムはこちら 10days_c ▶ day08-03 ▶ diary_renshuu.c

map.datをC:¥sourceディレクトリにコピーして実行すること。

【diary_renshuu.cより抜粋】

```
void viewData(char datafile[]) {
    diary walk_diary[31];
    diary tmp;
    char filename[15];
    char str[128];
    int i, j, file_c = 0;
    int c = 0; // 現時地点
    double total = 0, m_total = 0;

    dist_data map[100];                    ←最大数を変更
    dist_data tmp_map;                     ←新規追加
    int map_c = 0; //map最大添え字         ←数を変更

    FILE *fp; // 入力ファイルのファイルポインタ
    DIR *dir;
    struct dirent *dp;
```

```
    char *datafiles[100];
    double dist[31] = { 0.0 };   // 0で初期化する
    int lastday[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    //距離データの取得                        ←── この部分を追加
    if((fp = fopen("map.dat", "r")) == NULL) {
        printf("地域ファイルオープンエラー¥n");
        return;
    }
    while(fgets(str, sizeof(str), fp) != NULL) {
        sscanf(str, "%s %lf¥n",  &tmp_map.name, &tmp_map.dist);
        if(strlen(tmp_map.name) > 0) {
            strcpy(map[map_c].name, tmp_map.name);
            map[map_c++].dist = tmp_map.dist;
        }
    }
    fclose(fp);
    map_c--;

    //データファイル名一覧を取得する
    dir=opendir(datadir);
    (diary.cと同様なので略)
}
```

9
2
第９日
２時限目

A 解答プログラムはこちら 10days_c ▶ day09-02 ▶ 9-2_renshuu.c

【9-2_renshuu.cより抜粋】

```
void initBoard(void) {
    int i, j;
    int m, nokori;
    int r; //ランダムな値
    int digit[CMTX(MTX)-1]; //残り表示数値を記録する
    int count, k = 0;
    int tmp_board[CMTX(MTX)]; //表示数値を格納する

    do {
        nokori = CMTX(MTX)-1;
        for(i = 0; i < nokori; i++) { digit[i] = i+1; }
        k = 0;
        for(i = 0; i < CMTX(MTX)-1; i++, nokori--) {
            if(nokori > 1) { r = rand()%nokori; }
            else { r = 0; } //nokoriの数が1のときはdigit[0]
            tmp_board[i] = digit[r];// ボードに代入する
            for(m = r; m < nokori-1; m++) {
                digit[m] =  digit[m+1]; }
            if(tmp_board[i] != (i+1)) { k = 1; }
        }
    } while(k == 0); //揃っている状態なら揃えなおし
    tmp_board[CMTX(MTX)-1] = 0;

    count = 0;
    for(j = 1; j <= MTX; j++) { //縦方向の繰り返し
        for(i = 1; i <= MTX; i++) { //横方向の繰り返し
            board[j-1][i-1] = tmp_board[count];
            if(board[j-1][i-1] != 0) {
                printf(" %02d", tmp_board[count++]);
            }
        }
        printf("¥n");
    }
}
```

10-3
第10日
3時限目

marubatu_janken.cは変更なし。marubatu_board_renshuu.cとmarubatu_renshuu.cを作成し、marubatu_janken.c、marubatu_board_renshuu.c、marubatu_renshuu.cをコンパイルする。

A 解答プログラムはこちら　10days_c ▶ day10-03 ▶ marubatu_renshuu.c

marubatu.cをコピーしてmarubatu_renshuu.cを作成し、以下の変更を行う。
・外部変数p_i,p_jを削除
・自作関数writeCompMarkを呼び出した直後のprintf文を削除

A 解答プログラムはこちら　10days_c ▶ day10-03 ▶ marubatu_board_renshuu.c

marubatu_board.cをコピーしてmarubatu_board_renshuu.cを作成し、以下の変更を行う。
・compMarkCheck関数の定義とプロトタイプ宣言を追加
・writeCompMark関数を変更

【marubatu_board_renshuu.cより抜粋】

```
(宣言部分の冒頭は marubatu_board.c と同じなので略)

int compMarkCheck(int mode);

(marubatu_board.c と同じなので略)

/* コンピュータの選択した位置にマークをする関数 */
void writeCompMark(void) {
     int count = 0;          // 空いているマスの数
     int c_maru = MARU;      // コンピュータのマーク
     int i, j, r;
     int bmtx_i[100];
     int bmtx_j[100];
     int mark = 0;

     if(p_maru == MARU) { c_maru = BATU; }

     mark = compMarkCheck(1);
     if(mark == 0) { mark = compMarkCheck(2); }
     // ともにリーチはなかったのでランダムにマークする
     if(mark == 0) {
          for(j = 1; j <= MTX; j++) {
               for(i = 1; i <= MTX; i++) {
                    if(board[j-1][i-1] == 0) {
                         bmtx_i[count] = i;
```

```
                    bmtx_j[count] = j;
                    count++;
                }
            }
        }
        r = rand()%count;
        board[bmtx_j[r]-1][bmtx_i[r]-1] = c_maru;
        p_i = bmtx_i[r]; p_j = bmtx_j[r];
    }
    printf("コンピュータは・・・%c%d¥n", p_j+'a'-1, p_i);

}

/* リーチ状態の場所にマークをする関数
引数        1：コンピュータリーチチェック
            2：プレイヤーリーチチェック
戻り値      1：マークを行った
            0：マークをしていない
*/
int compMarkCheck(int mode) {
    int i,j;
    int player = 0, computer = 0;     // プレイヤー，コンピュータクのマーク数
    int b = 0;   // ブランクの位置
    int c_maru = MARU;   // コンピュータのマーク

    if(p_maru == MARU) { c_maru = BATU; }

    // 横方向
    for(j = 1; j <= MTX; j++) {
        // ○の数と×の数のチェック
        for(i = 1, player = 0, computer = 0, b = 0; i <= MTX; i++) {
            if(board[j-1][i-1] == p_maru) { player++; }
            else if(board[j-1][i-1] == c_maru) { computer++; }
            else { b = i; }
        }
        if(((mode == 2) && (player+1 == MTX && computer == 0)) ||
            ((mode == 1) && (computer+1 == MTX && player == 0))) {
            board[j-1][b-1] = c_maru;
            p_i = b; p_j = j;
            return 1;
        }
    }
    // 縦方向
```

```
        player = 0; computer = 0; b = 0;
        for(i = 1; i <= MTX; i++) {
            for(j = 1, player = 0, computer = 0, b = 0; j <= MTX; j++) {
                if(board[j-1][i-1] == p_maru) { player++; }
                else if(board[j-1][i-1] == c_maru) { computer++; }
                else { b = j; }
            }
            if(((mode == 2) && (player+1 == MTX && computer == 0)) ||
                ((mode == 1) && (computer+1 == MTX && player == 0))) {
                board[b-1][i-1] = c_maru;
                p_i = i; p_j = b;
                return 1;
            }
        }

        //ナナメ
        player = 0; computer = 0; b = 0;
        for(i = 1; i <= MTX; i++) {
            if(board[i-1][i-1] == p_maru) { player++; }
            else if(board[i-1][i-1] == c_maru) { computer++; }
            else { b = i; }
        }
        if(((mode == 2) && (player+1 == MTX && computer == 0)) ||
            ((mode == 1) && (computer+1 == MTX && player == 0))) {
            board[b-1][b-1] = c_maru;
            p_i = b; p_j = b;
            return 1;
        }
        player = 0; computer = 0; b = 0;
        for(j = 1; j <= MTX; j++) {
            if(board[j-1][MTX-j] == p_maru) { player++; }
            else if(board[j-1][MTX-j] == c_maru) { computer++; }
            else { b = j; }
        }
        if(((mode == 2) && (player+1 == MTX && computer == 0)) ||
            ((mode == 1) && (computer+1 == MTX && player == 0))) {
            board[b-1][MTX-b] = c_maru;
            p_i = b; p_j = MTX-b+1;
            return 1;
        }
        return 0;
}
```

（marubatu_board.c と同じなので略）

索引

記号

数字

A・B

C

D

E

F

翔泳社ecoProjectのご案内

株式会社 翔泳社では地球にやさしい本づくりを目指します。

近年、京都議定書の発効に伴って環境問題への関心が世界的な高まりを見せています。このような時代の要請に適応するためにも、企業は真剣に環境戦略を求められるようになってきました。業態の性格上、出版社は商品の生産ラインを所有してこなかったためか、環境問題を身近に感じることが今まで少なかったと言えます。しかし、事実上、大量の紙と原油系の化学物質製品を使用しているため、社会的責任を免れることはできません。

そこで、弊社では商品の制作工程において、環境への配慮を強化するために、エコロジー活動の一環として『翔泳社**ecoProject**』を立ち上げ、独自にエコロジー基準を設定しました（下表）。このうち4項目以上を満たしたものをエコロジー製品と位置づけ、シンボルマークをつけています。

このシンボルマークは葉をモチーフとしてデザインされています。
木から抽出されたパルプでつくられる紙。それを原料とする本。
本のもとは木なのです。

環境を考慮した技術で生産された本が適正にリサイクルされる。
そのことで新しい緑が育まれる。
はじめは小さな小さな葉っぱのような活動が徐々に枝葉を広げ、
一本の大樹となるように願っています。
そんな思いがつまったマークです。

ecoProject

資材	基準	期待される効果	本書採用
装丁用紙	無塩素漂白パルプ使用紙 あるいは 再生循環資源を利用した紙	有毒な有機塩素化合物発生の軽減（無塩素漂白パルプ） 資源の再生循環促進（再生循環資源紙）	○
本文用紙	材料の一部に無塩素漂白パルプ あるいは 古紙を利用	有毒な有機塩素化合物発生の軽減（無塩素漂白パルプ） ごみ減量・資源の有効活用（再生紙）	○
製版	CTP（フィルムを介さずデータから直接プレートを作製する方法）	枯渇資源（原油）の保護、産業廃棄物排出量の減少	○
印刷インキ*	植物油を含んだインキ	枯渇資源（原油）の保護、生産可能な農業資源の有効利用	○
製本メルト	難細裂化ホットメルト	細裂化しないために再生紙生産時に不純物としての回収が容易	○
装丁加工	植物性樹脂フィルムを使用した加工 あるいは フィルム無使用加工	枯渇資源（原油）の保護、生産可能な農業資源の有効利用	○

*：パール、メタリック、蛍光インキを除く

著者紹介
◎
坂下 夕里
さかした ゆうり

大学卒業後、IT企業にてシステム開発に従事。退社後、知人経由でちらほらとWebシステム開発の仕事を受けはじめ、気がついたらいつのまにかフリーに。
著書に、『10日でおぼえるC言語入門教室』『Perl/CGI辞典』『これならわかるトレーニングドリルC』『これならわかるトレーニングドリルJava』『これならわかるJava入門の入門 第2版』。

「10日でおぼえる」のホームページ

本書の内容確認には十分気を付けていますが、万一プログラムコードや本文の誤記などで訂正が発生した場合には、下記ホームページにて訂正内容をお知らせいたします。ご利用ください。

http://www.shoeisha.com/book/hp/10days/

ブックデザイン　株式会社アレフ・ゼロ（宇田 俊彦／鈴木 住枝）
DTPオペレーション　有限会社テクスト

10日(とおか)でおぼえる C言語入門教室(シーげんごにゅうもんきょうしつ) 第2版(だいにはん)

2009年8月6日　初版第1刷発行

●
著者　坂下 夕里
発行人　佐々木 幹夫
発行所　株式会社翔泳社
(http://www.shoeisha.co.jp/)
印刷・製本　日経印刷株式会社
●

ISBN978-4-7981-1905-2　Printed in Japan

10日でおぼえる

C言語
入門教室
第2版

10日でおぼえる C言語入門教室 第2版

坂下夕里 著

SE
SHOEISHA

●●カバー写真
『ビルディングフォーム』

ISBN978-4-7981-1905-2
C3055 ¥2200E

株式会社翔泳社
定価：本体2,200円+税

9784798119052

1923055022003

10日でおぼえる C言語入門教室 第2版

坂下夕里 著

SE SHOEISHA